Book Recommendation System — Collaborative & Content-Based Filtering Approaches

Milos Saric

GitHub LinkedIn saricmilos.com Real Skills Over Degrees

November 20, 2025

Abstract

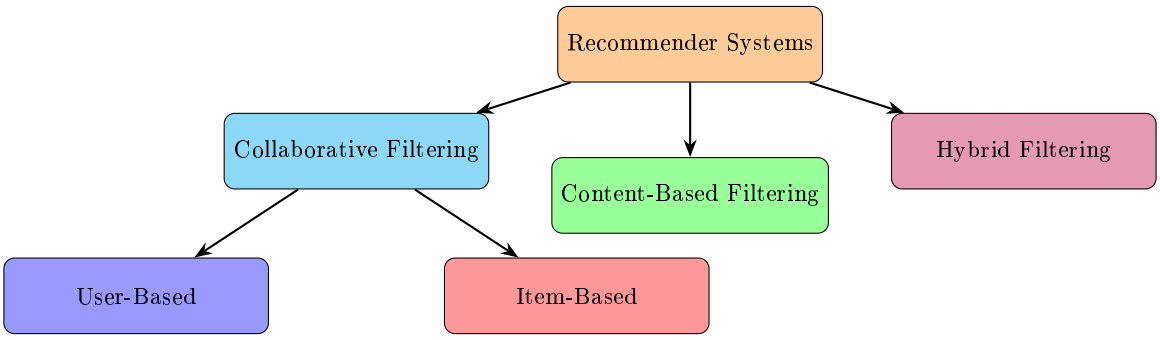
This report investigates the Kaggle Book Recommendation dataset to develop intelligent book recommendation systems using both collaborative filtering and content-based methods. A key application of machine learning is generating personalized recommendations for users, aiming to boost revenue, engagement, or other performance metrics. As a result, understanding how these recommendation algorithms work is an essential skill for any Machine Learning Engineer. Recommender systems have become a vital component of online platforms such as YouTube, Amazon, and Netflix, helping users discover content tailored to their preferences. By analyzing user behavior and item attributes, these systems predict user interests, enhancing user experience while driving engagement and revenue. This study highlights the implementation, advantages, and practical significance of recommendation algorithms in modern digital platforms.

---

1 Introduction and Problem Definition

Recommender systems aim to predict user preferences and suggest items they are most likely to enjoy. This project focuses on developing a **book recommendation system** using the Kaggle **Book-Crossing Dataset**, leveraging both **collaborative**, **content-based approaches** and **hybrid filtering approaches** to deliver personalized book suggestions. The goal is to design a system that accurately predicts and recommends books a user is likely to enjoy based on past interactions, ratings, and preferences. A clear understanding of this problem ensures that all subsequent analysis and modeling efforts align with the primary objective.

The main purpose of a recommender system is to boost product sales. Businesses use these systems to maximize their profits by suggesting products that customers are likely to be interested in. By highlighting relevant and appealing items, recommender systems help users discover products they might otherwise miss — ultimately driving higher sales and greater profits for the company.



1.1 Scope

The analysis focuses exclusively on the Kaggle dataset, which includes:

- **Users:** demographic and identification data.
- **Books:** metadata such as title, author, year, publisher, and cover images.
- **Ratings:** explicit ratings (1–10) and implicit feedback (0 for interactions without ratings).

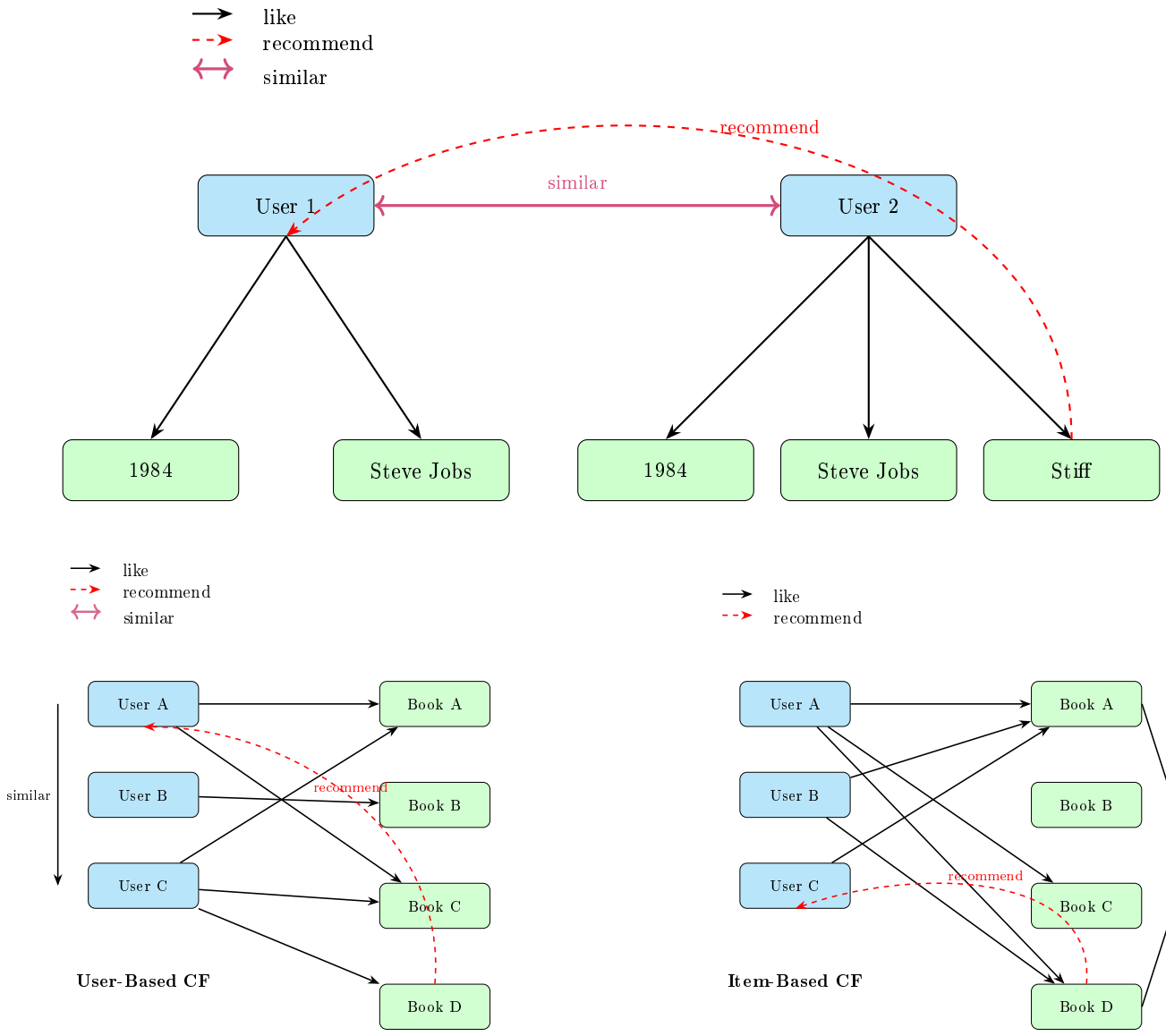
Predictions are restricted to this dataset without using external sources unless explicitly integrated in advanced phases.

1.2 Stakeholders

- **Readers / Users:** receive personalized book recommendations.
- **Publishers / Authors:** gain insights into reader interests for better targeting.
- **Data Scientists / ML Practitioners:** test and optimize recommendation algorithms.
- **Platform Developers / Businesses:** improve user engagement, retention, and revenue through effective recommendations.

2 Collaborative Filtering

Collaborative filtering creates a model based on a user's past actions such as items they have purchased, selected, or rated, as well as the behavior of other users with similar preferences. This model is then used to recommend items or predict ratings for items that the user is likely to be interested in. Collaborative filtering is a recommendation method that predicts a users preferences based on the choices of other users with similar tastes. It relies on both user and item information, often organized in a user-item matrix.
This approach is widely used in industry across platforms like YouTube, Netflix and Amazon. For example, if users with similar past actions/tastes buy a product on Amazon, the system can suggest that same product to you. YouTube leverages this technique to recommend videos, Amazon uses it to suggest books and products, and Netflix relies heavily on collaborative filtering to personalize movie recommendations.



Collaborative filtering relies heavily on user data. As users continue to browse, purchase, and rate products, their preferences evolve over time. Because of this, the underlying recommendation model must be regularly updated and refined to stay accurate and relevant. In practice, this means continuously integrating new data and adjusting the system to reflect changing user behavior and market trends.
The two main sources of data for collaborative filtering are:

- **Explicit data:** Information that users actively provide, such as ratings, reviews, or responses to surveys and questionnaires.
- **Implicit data:** Information inferred from user behavior, including browsing history, clicks, time spent on content, likes, shares, and other engagement signals.

Both types of data are essential for understanding user preferences. Explicit ratings capture direct feedback, while implicit data reveals natural behavioral patterns that help improve recommendation accuracy over time.

2.1 User-Based Collaborative Filtering

User-based collaborative filtering recommends items to a user by identifying other users with similar preferences. If a similar user liked an item, it can be suggested. In our user based book recommendation system, we begin by constructing a matrix with users as rows and books (identified by ISBN) as columns let's call this the user-book matrix. Each entry in the matrix represents either a rating from 1 to 10, indicating that the user has read and rated the book, or a zero if the user has not interacted with that book. For example, if user J has rated book K, the corresponding cell contains that rating. If user J has not read or rated book K, the cell is simply zero.
User-based collaborative filtering recommends items to a user by identifying other users with similar preferences. If a similar user liked an item, it can be suggested.
Similarity between users can be measured using metrics such as **Cosine similarity**, **Pearson correlation**, or **Euclidean distance**. Read more about similarity at Section 7.

2.1.1 Algorithmic Workflow

1. Represent each user as a vector in item space:

   In user-based collaborative filtering, we start from a set of observed user-item ratings

   $$R = \{(u, i, r_{ui}) \mid u \in U,\, i \in I\} \tag{1}$$

   where U is the set of users, I is the set of items (books), and r_{ui} is the rating given by user u to item i.

   We represent this data as a **user-item rating matrix** $M \in \mathbb{R}^{|U| \times |I|}$:

   $$M_{ui} = \begin{cases} r_{ui}, & \text{if user } u \text{ rated item } i \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

   Each **row** of this matrix corresponds to a *user vector*:

   $$\mathbf{u}_k = [\, r_{k1}, r_{k2}, \ldots, r_{k|I|} \,] \in \mathbb{R}^{|I|} \tag{3}$$

   where r_{kj} is the rating of user k for item j, and I is the number of items. Thus, the moment we pivot our data from a triplet (u, i, r_{ui}) format into a matrix form, we have implicitly constructed a high-dimensional vector representation for every user.

2. Compute user-user similarity, e.g., using **cosine similarity**:

   $$\text{sim}(u, v) = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\|\, \|\mathbf{v}\|} \tag{4}$$

   where $\mathbf{u} \cdot \mathbf{v}$ is the dot product, and $\|\mathbf{u}\|$ is the Euclidean norm:

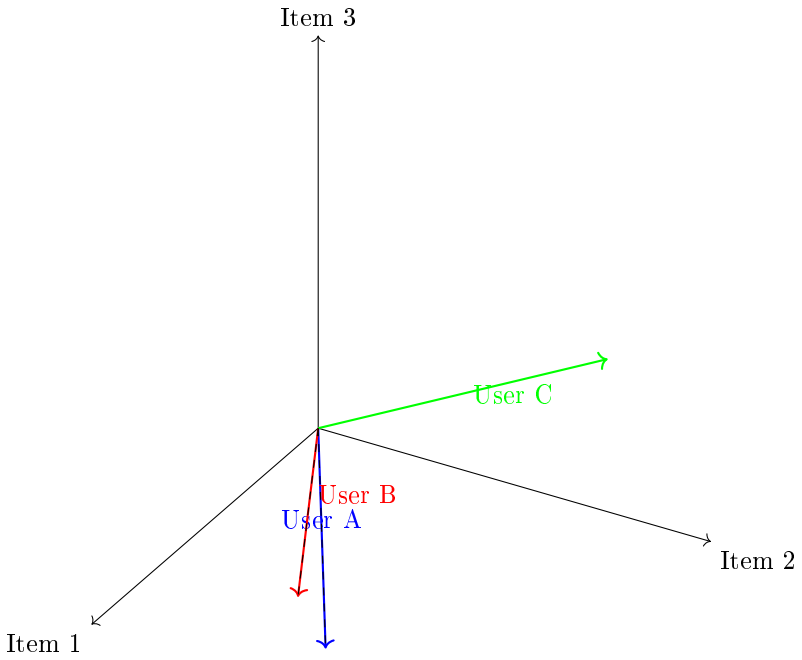
   $$\|\mathbf{u}\| = \sqrt{\sum_{j=1}^{M} r_{uj}^2} \tag{5}$$

   - Each user is a vector in M-dimensional space, where each axis represents an item.

- The length of the vector corresponds to the magnitude of their ratings.
- Cosine similarity measures the **angle between user vectors**, i.e., the similarity in rating patterns.
- Example:

$$\text{User A} = [5, 3, 0], \quad \text{User B} = [4, 2, 0]$$

Even though User B gives slightly lower ratings, the *pattern is similar*, resulting in a high cosine similarity.



Cosine similarity compares rating patterns, not absolute rating levels. Length of the vector = total "strength" of user ratings, but cosine similarity is normalized so that only direction matters.

3. Predict ratings for a user u on item i using top-K similar users $N(u)$:

There are two related but different formulas commonly used for user-based collaborative filtering (CF).

The correct and most widely used formula for predicting a user's rating is:

$$\hat{r}_{ui} = \bar{r}_u + \frac{\sum_{v \in N(u)} \text{sim}(u, v) \cdot (r_{vi} - \bar{r}_v)}{\sum_{v \in N(u)} |\text{sim}(u, v)|} \tag{6}$$

- $\hat{r}_{ui}$ is the *predicted rating* of user u for item i.
- $\bar{r}_u$ represents the *average rating* of user u (the user's rating bias).
- $N(u)$ denotes the set of *nearest neighbors* (similar users to u) who have rated item i.
- r_{vi} is the *rating given by neighbor* v to item i.
- $\bar{r}_v$ is the *average rating* of user v.
- $\text{sim}(u, v)$ measures the *similarity* between users u and v, often computed using cosine similarity, Pearson correlation, or other metrics.

A simpler, less accurate version of the formula is sometimes used:

$$\hat{r}_{ui} = \frac{\sum_{v \in N(u)} s(u, v) \cdot r_{vi}}{\sum_{v \in N(u)} |s(u, v)|} \tag{7}$$

- This version ignores user rating biases ($\bar{r}_u$ and $\bar{r}_v$).
- It assumes all users rate on the same scale, which is often not true in practice.

- Although simpler, it generally performs worse in predicting ratings accurately.

In summary, the mean-centered formula is preferred for more accurate and bias-adjusted predictions, while the simpler one can be used as a baseline or when user averages are unavailable.

4. Evaluation of Collaborative Filtering Predictions:

   Evaluating the performance of a collaborative filtering (CF) model involves measuring how accurately the system predicts user ratings and how effective its recommendations are. Two major evaluation perspectives are commonly used: prediction accuracy and recommendation quality.

5. **Evaluation Procedure**

   A standard workflow:

   (a) Split the dataset into training and test sets (e.g., 80/20 split).

   (b) Build the CF model using the training set.

   (c) Predict ratings $\hat{r}_{ui}$ for test set user–item pairs.

   (d) Compare predicted and actual ratings using MAE, RMSE, etc.

   (e) Optionally, evaluate top-N recommendations using Precision@N and Recall@N.

2.1.2 Summary

- Each user is a vector in an $|I|$-dimensional space (items are dimensions).
- Ratings are vector components (coordinate values).
- Similarities are computed between users' vectors.
- Predictions are weighted averages of neighbors' ratings.

2.2 Item-Based Collaborative Filtering

Item-based collaborative filtering recommends items to a user by analyzing similarities between items rather than users, based on user interactions. The intuition is: if a user liked a particular item, they are likely to enjoy other items that received similar ratings from other users. For example, in a book recommendation system, if two books are rated similarly by many users, they are considered similar, and a user who liked one might like the other.

Matrix Representation The cosine similarity function itself does not know whether we are computing user-based or item-based similarity. The distinction depends solely on the orientation of the rating matrix M. We have two perspectives:

1. **User-Based Filtering:** Each user is represented as a vector in the *item space*. This yields a user–user similarity matrix of shape $|U| \times |U|$.
2. **Item-Based Filtering:** Each item is represented as a vector in the *user space*. This yields an item–item similarity matrix of shape $|I| \times |I|$.

In practice, this difference is implemented by transposing the matrix before applying cosine similarity:

$$\text{user-based: cosine\_similarity}(M), \quad \text{item-based: cosine\_similarity}(M^T).$$

Each item becomes a vector in *user space*, with dimensions corresponding to users:

$$\mathbf{i}_j = [r_{1j}, r_{2j}, r_{3j}, \ldots, r_{Nj}] \in \mathbb{R}^{|U|} \tag{8}$$

where r_{uj} is the rating given by user u to item j, and $N = |U|$ is the number of users. The transposition means that the same rating matrix M can be used, but similarity computations are performed along the columns instead of rows.

Item Similarity Similar to user-based CF, we compute the similarity between item vectors using metrics like cosine similarity:

$$\text{sim}(i,j) = \frac{\mathbf{i}_i \cdot \mathbf{i}_j}{\|\mathbf{i}_i\| \, \|\mathbf{i}_j\|} \tag{9}$$

This produces an $|I| \times |I|$ item-item similarity matrix. The similarity values are then used to predict a user's rating for a target item based on the ratings of similar items they have already rated.

Prediction Formula The predicted rating of user u for item i is computed as:

$$\hat{r}_{ui} = \frac{\sum_{j \in N(i)} s(i,j) \cdot r_{uj}}{\sum_{j \in N(i)} |s(i,j)|} \tag{10}$$

where:

- $N(i)$ is the set of top-K items similar to i that user u has rated.
- $s(i,j)$ is the similarity between items i and j.
- r_{uj} is the rating of user u for item j.

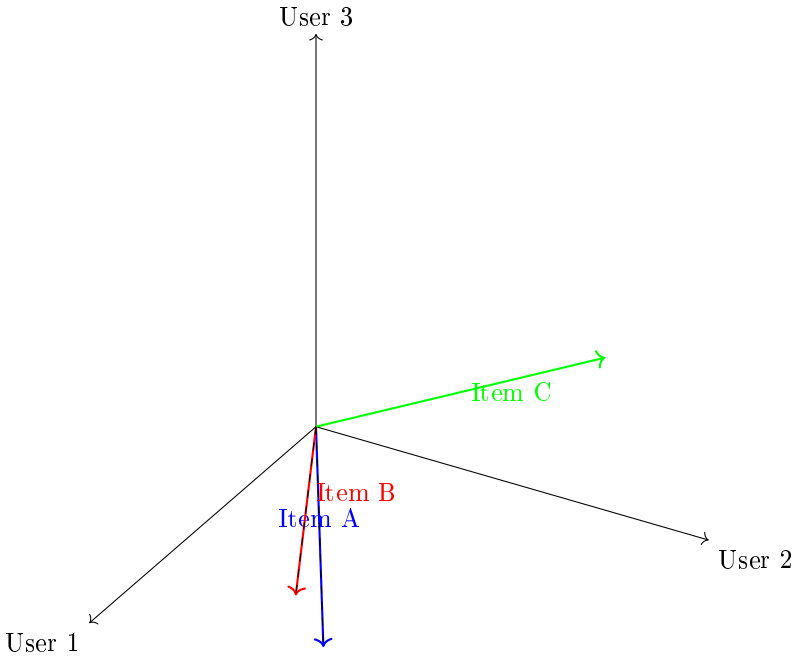
This formula is analogous to the user-based CF prediction, but here the weights come from item-item similarity rather than user-user similarity.

Algorithmic Workflow

1. Represent each item as a vector in user space ($|U|$-dimensional).
2. Compute the item-item similarity matrix using cosine similarity.
3. For each user and target item, identify the set of similar items the user has rated ($N(i)$).
4. Predict the rating using a weighted sum of the user's ratings on similar items.
5. Optionally, generate top-N recommendations based on predicted ratings.

Intuition and Advantages

- Each item is a vector in user space, analogous to users being vectors in item space for user-based CF.
- Cosine similarity measures which items are rated similarly across users.
- IBCF often performs better in sparse datasets, as items tend to have more ratings than individual users, making similarity computations more robust.
- Reduces computational cost when the number of items is smaller than the number of users.



3 Content Based Filtering

Content-Based Filtering recommends items that are *similar to those a user has previously liked.* It relies on the features or attributes of the items, such as movie genre, product category, author, or keywords. The system first builds a user profile based on the characteristics of items the user interacted with, then suggests other items with similar properties. It generates recommendations by analyzing the features of items and matching them to a user's preferences. Essentially, the system compares a **user profile** with an **item profile** to predict which items the user is likely to engage with. Content-Based Recommender Systems often build a personalized model for each user. The process begins by collecting the features of items the user has interacted with which forms the user profile. These items serve as a training set for a user-specific classifier or regression model.
In the model, the item features act as independent variables, while the user's past behavior-such as ratings, likes, or purchases serve as the dependent variable. Once trained, the model predicts the user's likely response to new items, allowing the system to recommend items that align with the user's preferences. The **item profile** captures the characteristics of an item, including structured features or descriptive metadata. For example, a streaming service may represent movies using attributes such as genre, director, release year, or cast.
The **user profile** reflects the individual's preferences and past behavior. It is typically built from items the user has previously interacted with, including ratings, likes, dislikes, searches, or other engagement data. In short, this approach focuses on *what the user likes*, rather than what other users like.

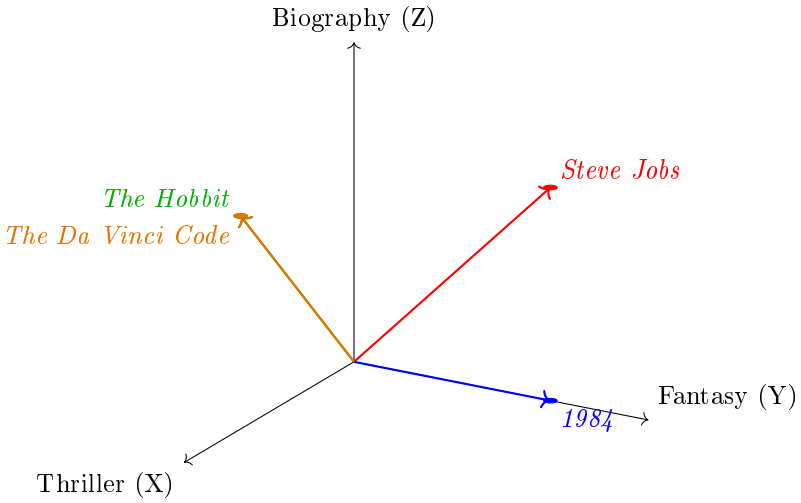
3.1 Item Representations in Content-Based Filtering

In content-based filtering, items and users are often represented as vectors in a multi-dimensional feature space. Each item's feature such as genre, author, or other metadata define its coordinates. Boolean values (1 for presence, 0 for absence) are commonly used to indicate whether an item possesses a particular feature.
Example: A few novels represented by three genres (Biography, Fantasy, Thriller):

Table 1: Boolean Feature Representation of Novels

Novel	**Biography**	**Fantasy**	**Thriller**
Steve Jobs	0	1	1
1984	0	1	0
The Hobbit	1	0	1
The Da Vinci Code	1	0	1

In this 3D vector space, each dimension corresponds to a feature (Adventure, Bildungsroman, Children). A novel's coordinates are determined by its Boolean values. For example:



The closer two novels are in this space, the more similar they are according to the selected features. For instance, *The Hobbit* and *The Da Vinci Code* overlap completely, while *Steve Jobs* is closer to *1984* than to the thriller novels. Since The Hobbit and The Da Vinci Code are very similar in the feature space, a user who has previously purchased The Hobbit is likely to be recommended The Da Vinci Code, as it closely matches their interests.

Visualization: We can plot these items in a 3D Cartesian coordinate system, with each axis representing one genre. Novel vectors are points in this space, and proximity reflects similarity. Adding more features (e.g., Fantasy, Gothic) increases the dimensionality, adjusting item positions accordingly.

In content-based filtering, each user is represented by a vector that summarizes their preferences. This user vector, denoted $\mathbf{u}_x$, is typically computed as a weighted average of the vectors of the books the user has liked or rated highly:

$$\mathbf{u}_x = \frac{1}{N_x} \sum_{i \in I_x} w_{xi} \cdot \mathbf{v}_i \tag{11}$$

where I_x is the set of books user x has interacted with, w_{xi} is the weight associated with book i for this user (such as a normalized rating or binary like/dislike), $\mathbf{v}_i$ is the feature vector of book i, and N_x is a normalization factor.

Once the user vector is constructed, the next step is to compute the similarity between this user vector and the vector of every book in the catalog. The most common similarity measure is cosine similarity, defined as

$$\text{sim}(\mathbf{u}_x, \mathbf{v}_i) = \frac{\mathbf{u}_x \cdot \mathbf{v}_i}{\|\mathbf{u}_x\| \, \|\mathbf{v}_i\|} \tag{12}$$

High similarity indicates that the book aligns closely with the user's preferences, making it a strong candidate for recommendation. Low similarity suggests the book is less likely to match the user's interests. After computing the similarity for all books, they are sorted by score, and the top-K books are recommended to the user, excluding those they have already rated. This process allows the system to suggest books that are most likely to match the user's tastes based on the features encoded in the vectors.

3.2 Item-Based Collaborative Filtering vs. Content-Based Filtering

Although both methods aim to recommend items similar to those a user already liked, they differ in what information they rely on and how they define similarity.

Item-Based Collaborative Filtering (CF) focuses on *user behavior patterns*. Items are considered similar if many users have interacted with both of them in similar ways. For example, if several users who bought *Book A* also bought *Book B*, the system concludes that these two books are related, even if their topics are entirely different. This method depends on user ratings or implicit feedback (such as clicks or purchases) to measure similarity between items. In essence, the approach follows the idea: *"Users who liked this item also liked that item."*

Content-Based Filtering (CBF), on the other hand, focuses on *the characteristics of the items themselves*. It recommends new items similar to those the same user has already liked, based on item features such as genre, keywords, author, or description. For instance, if a user enjoys a mystery novel with a strong female lead, the system suggests other mystery novels with similar themes or features. The guiding principle is: *"You liked this because of its content, so you might like similar content."*

3.3 Advantages and Disadvantages of Collaborative and Content-Based Filtering

Collaborative Filtering: Advantages: Collaborative filtering can uncover new items that a user may not have considered before by leveraging the preferences of similar users. This allows for more diverse recommendations that go beyond the user's past interactions.
Disadvantages: Collaborative filtering suffers from the *cold-start problem*. New users have no interaction history, making it difficult to find similar users, and new items lack sufficient ratings to be recommended effectively. Additionally, calculating similarities between users can be computationally expensive for large datasets.
Content-Based Filtering: Advantages: Content-based filtering handles new items efficiently, as recommendations rely on item features rather than prior user interactions. It also provides greater transparency, explaining why a particular item is recommended. For example, a movie may be suggested because it shares the same genre or actors as previously liked movies, allowing users to make informed decisions.
Disadvantages: Content-based filtering is limited by the features available to describe items. If a user's preference is based on characteristics not captured in the item profile (e.g., specific plot elements or production details), the system may fail to provide relevant recommendations. Moreover, it tends to overspecialize, suggesting items very similar to those already liked, which reduces diversity and limits discovery of new or unexpected items.

4 Exploratory Data Analysis (EDA) and Data Preparation

We begin by loading our datasets and cleaning column names for consistency. We remove leading/trailing spaces, convert all names to lowercase, and replace dashes (-) with underscores (_). This makes the DataFrame easier to work with in subsequent analysis.

4.1 Books Dataframe

Our books dataset contains several key columns: **isbn** (unique book identifier), **book_title** (title of the book), **book_author** (author name), **year_of_publication** (publication year), **publisher** (publisher name), and **image_url_s/m/l** (small, medium, and large book cover images). For analysis, the image columns (`image_url_s`, `image_url_m`, `image_url_l`) are dropped as they are not needed. All remaining attributes are of type `object`.
The dataset contains **271,360 book entries**. Key observations include:

- `isbn` Every book has a unique ISBN, making it a perfect **unique identifier**.
- `book_title` There are **242,135 unique titles** (about **89%** of entries). Most titles are unique, though some duplicates exist, likely due to *different editions or reissues*. For example, *Selected Poems* appears 27 times with different ISBNs.
- `book_author` Around **102,022 unique authors** are present, meaning each author has, on average, about **2–3 books**.
- `publisher` Only **16,807 unique publishers** appear (roughly **6%** of entries), which is expected since many books come from the same publisher. *Harlequin* is the most common.
- `year_of_publication` There are just **202 distinct years**, typical because many books share the same publication year. **2002** is the most frequent year, appearing in 13,903 books.

Overall, the dataset is consistent: **ISBNs are unique**, while other fields naturally repeat due to multiple books by the same authors, publishers, or publication years.

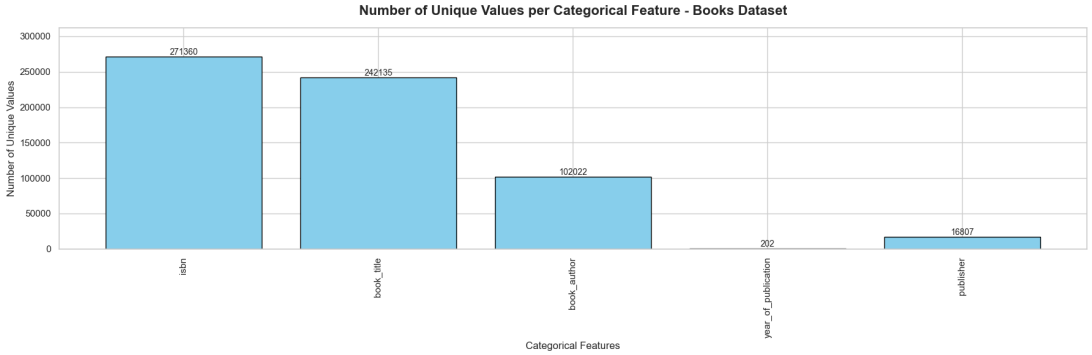


Figure 1: Unique Values Books DataFrame

All book related columns are mostly complete. The book author and publisher columns each have only 2 missing values out of 271,360 entries, while all other columns are fully populated.
There are two books with missing authors: "A+ Quiz Masters:01 Earth" (ISBN 0751352497) and "The Credit Suisse Guide to Managing Your Personal Finances" (ISBN 9627982032). We looked up the correct author information online and filled them as "Dorling Kindersley Publishing Staff" and "Credit Suisse," respectively.
Similarly, two books had missing publishers: "Tyrant Moon" (ISBN 193169656X) and "Finders Keepers" (ISBN 1931696993). We searched for the correct publishers and updated them to "Mundania Press LLC" and "Random House Publishing Group."
After these corrections, all books now have complete information for both authors and publishers.
We further explored the dataset by plotting value counts and found that the authors with the most published books are Agatha Christie (632), William Shakespeare (567), Stephen King (524), Ann M. Martin (423), and Carolyn Keene (373). The publishers with the most books are Harlequin (7,535), Silhouette (4,220), Pocket (3,905), and Ballantine Books (3,783).

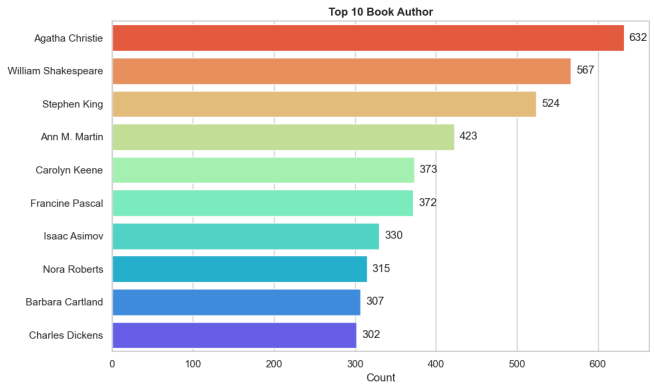


Figure 2: Author with most published books

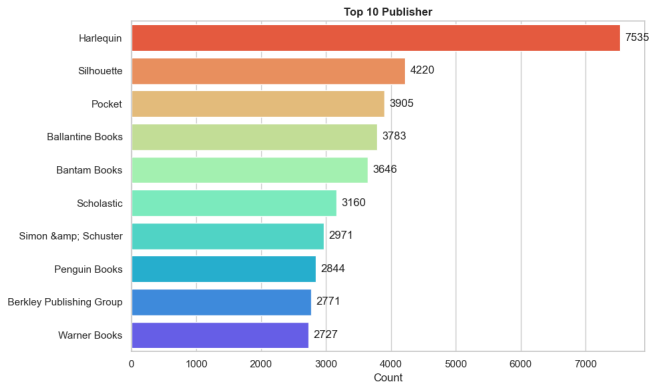


Figure 3: Publisher with most published books

The `year_of_publication` column is quite messy, containing a mix of valid and invalid entries. It includes integers like 1999 or 2002, strings that represent years such as "2000" or "1998", non year strings like publisher names ("DK Publishing Inc", "Gallimard"), and impossible or unrealistic years such as 0, 1378, 1806, 2030, and 2050.
Interestingly, only a few rows contain genuinely non-numeric entries, even though there are over 65,000 string values. Most of these strings are actually numeric ("1998", "2003", "0") and will need to be converted to integers and cleaned before analysis.
To clean this column, we can convert all values to numeric using `pd.to_numeric(errors='coerce')`, which turns invalid entries into `NaN`. Then, we replace unrealistic years (less than 1000 or greater than 2025) with `NaN`, fill missing values with the median, and finally convert the column to integers. This ensures that `year_of_publication` s consistent, numeric, and ready for analysis.

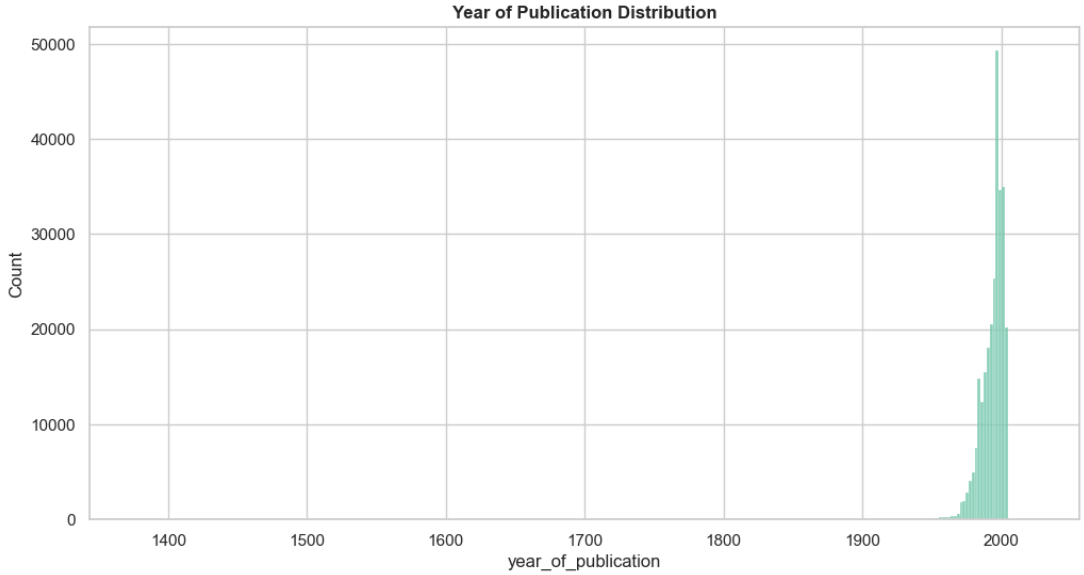


Figure 4: Histogram of Publication age

We can also notice from the histogram that very few books in our dataset were published before 1900, so we can remove those values and keep only books published from 1900 onwards.

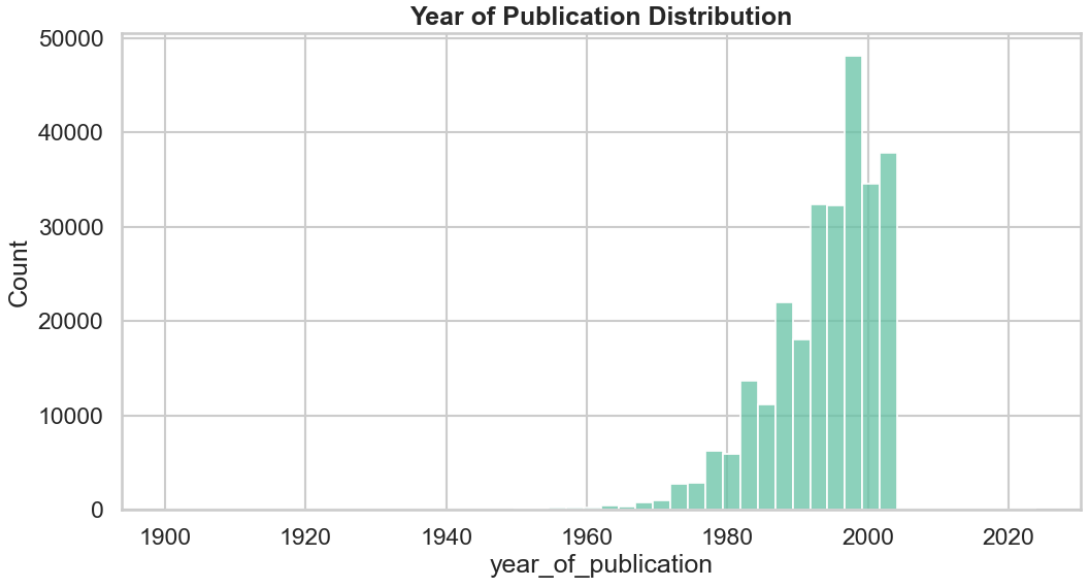


Figure 5: Histogram of Publication age from 1900

4.2 Ratings Dataframe

Our ratings dataset contains three columns: **user_id**, **isbn**, and **book_rating**, with approximately 1,149,780 entries. There are no missing values. The **user_id** column is just an identifier, so statistics like mean or std are not meaningful. The **book_rating** column ranges from 0 (book read but not rated) to 10, with a mean of 2.87 and a standard deviation of 3.85, indicating high variability. About 50% of ratings are 0 and 75% are below 7, showing a sparse ratings matrix typical of recommendation datasets.
Data types: *isbn* is categorical, *user_id* and *book_rating* are integers. **Unique values:** isbn: 340,556 (∼30%), user_id: 105,283 (∼9%), book_rating: 11 (0–10).

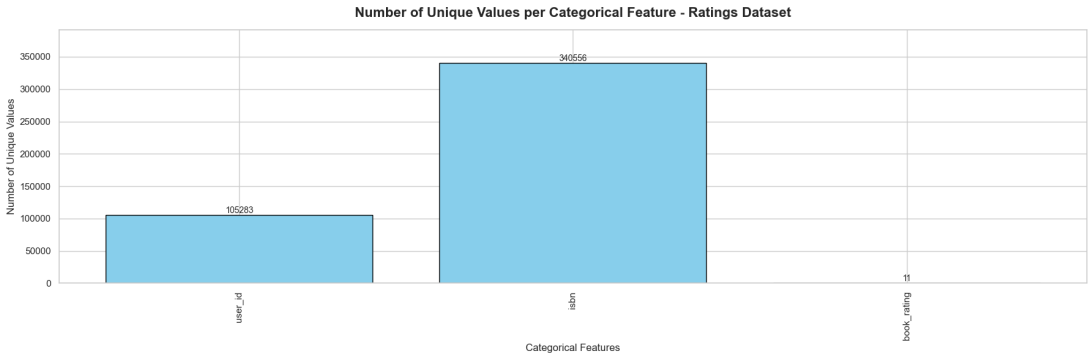


Figure 6: Unique Values Ratings DataFrame

Many books were rated by multiple users, and many users rated multiple books.
The histogram shows the distribution of user ratings for books. A large number of users (716,109) did not provide a rating, which reflects real life behavior, most people enjoy a product and move on without leaving an online review. While this is natural, it poses a challenge for collaborative recommendation systems, as unrated items result in zero-length vectors and sparse data. For user-based filtering, new users with few or no ratings make it hard to find similar users. For item-based filtering, new books with few ratings are difficult to recommend, leading to the cold start problem and sparse data issues.

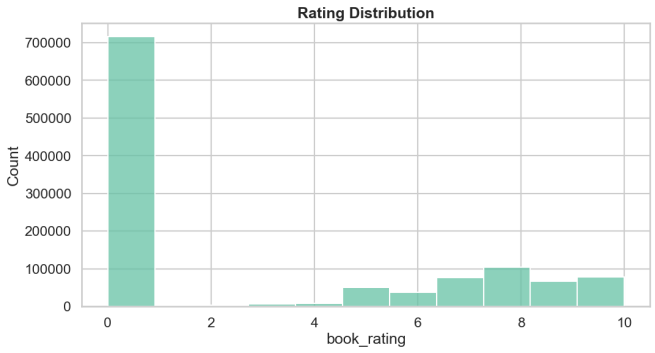


Figure 7: Histogram of Ratings given

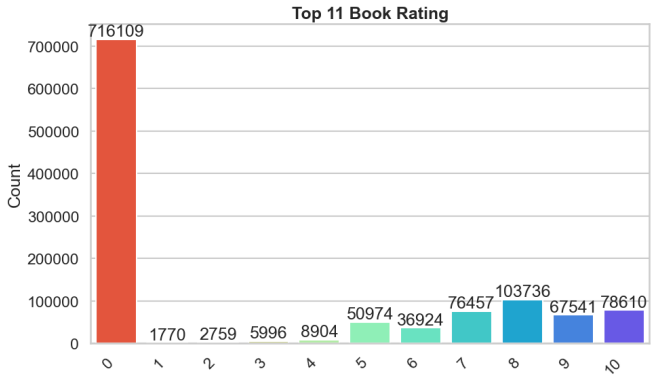


Figure 8: Histogram of Ratings given with values

Our ratings dataset contains no missing values.

4.3 Users Dataframe

We start by preprocessing the `location` column in the user dataset. User-entered locations are often messy: typos, alternate spellings, subregions, or single-entry countries. Our goal is to standardize and simplify these values.

- **split_location(df):** Splits the `location` column into `city`, `state`, and `country`, trims whitespace, and fills missing values with `'unknown'`.
- **clean_country(df, country_mapping, region_mapping):** Standardizes country names, groups rare countries into `'other'`, removes quotes/empty strings, maps each country to a `region`, and drops the original `country` column.
- **clean_city(df, top_n=50):** Cleans city names, replaces invalid/missing entries with `'unknown'`, keeps only the top N frequent cities, labels the rest as `'other'`, and drops the original column.
- **clean_state(df, top_n=50):** Similar to `clean_city`, but for states.
- **preprocess_location(df, ...):** Runs all the above steps and returns cleaned columns: `city_clean`, `state_clean`, `country_clean`, and `region`.

Overall, this process transforms messy location data into standardized, manageable categories suitable for analysis or modeling.
The dataset contains **278,858 users**, but only **168,096 have a recorded age**, meaning roughly **110,000 users did not provide their age**.
The `user_id` column is fully unique and serves as a reliable identifier. Its average value (~139,429) is not meaningful.
The `age` column has an average of approximately **35 years** with a standard deviation of about **14 years**, indicating a fairly wide spread.
Examining the age distribution:

- The **youngest recorded age is 0**, which is clearly invalid or missing.
- **25% of users are younger than 24**, and **50% are younger than 32** (median).
- **75% are younger than 44**, meaning most users fall between their mid-20s and mid-40s.
- The **maximum recorded age is 244**, an obvious outlier or error.

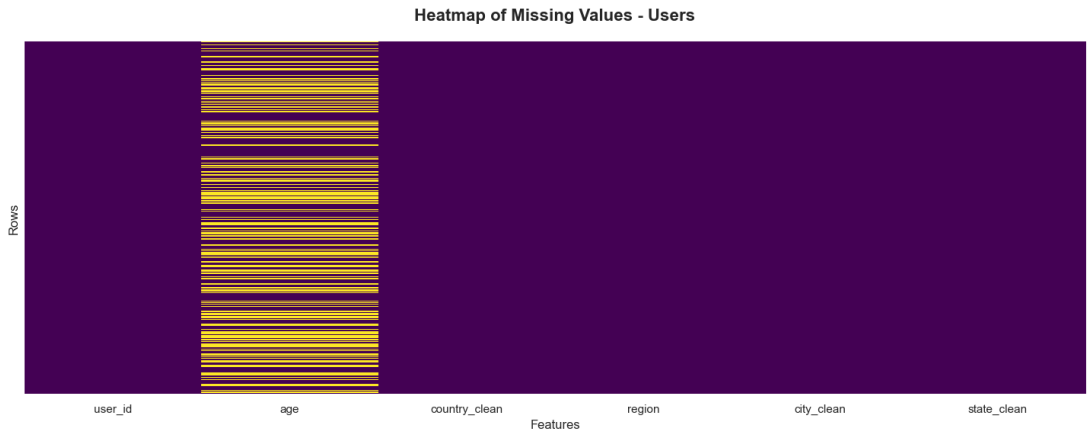


Figure 9: Missing Values Users DataFrame

Regarding column uniqueness:

- `user_id` is fully unique (100%), serving as a unique identifier.
- `age` has 165 unique values (~0.06%), highlighting the presence of outliers.
- `country_clean` has 131 unique entries (~0.05%), `city_clean` 51 (~0.02%), and `state_clean` 50 (~0.02%).
- `region` is the least varied, with only 13 unique values (~0.00%).

Overall, the age data contains some missing and unrealistic values, but most users are in the **24 44 year range**, and all other columns are largely complete and consistent.

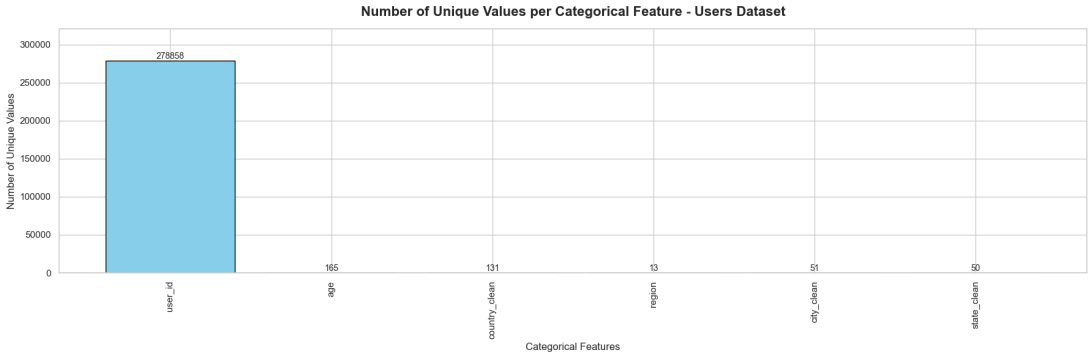


Figure 10: Unique Values Users DataFrame

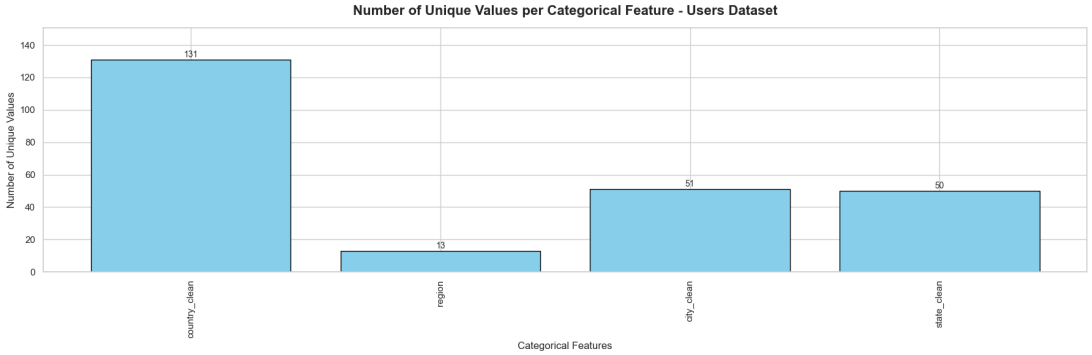
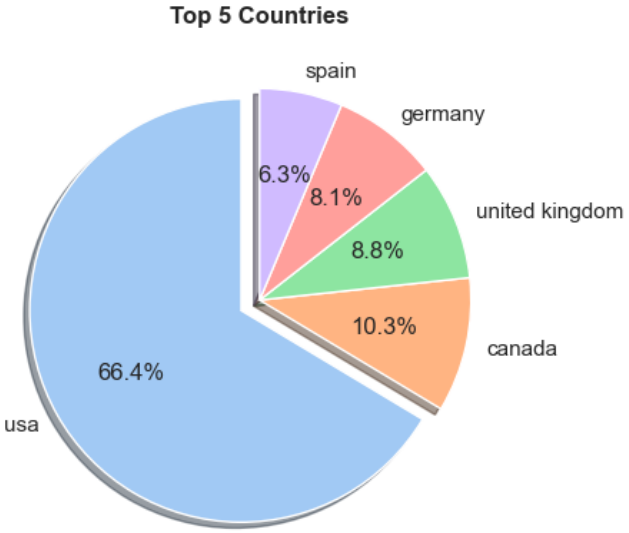


Figure 11: Unique Values Without ID Users DataFrame

Most reader ratings come the USA leading at 139,599 ratings, followed by Canada (21,622), the United Kingdom (18,587), Germany (17,092), and Spain (13,313).



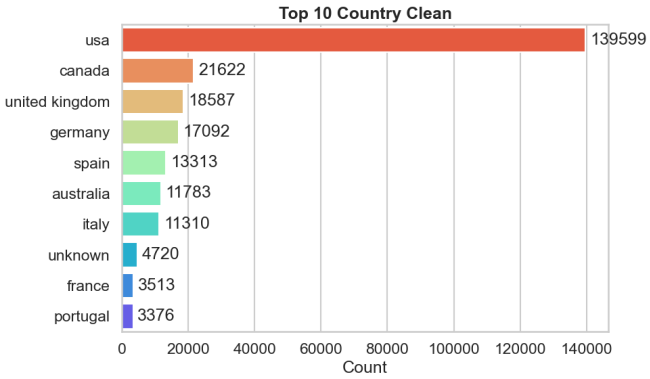


Figure 13: Histogram of countries of user's who gave ratings

Figure 12: Piechart of countries of user's who gave ratings

Most users are grouped under the category 'Other' because, during preprocessing, all less-popular cities were combined into this category. Excluding 'Other', we can see that major cities dominate user ratings, with London (4,105), Barcelona (2,664), Toronto (2,342), Madrid (1,933), Sydney (1,884), Chicago (1,566), and New York (1,445) leading the list.

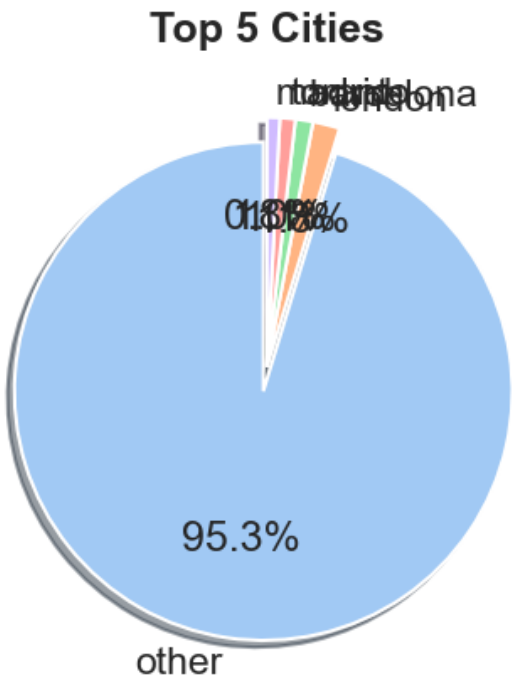


Figure 14: Piechart of cities of user's who gave ratings

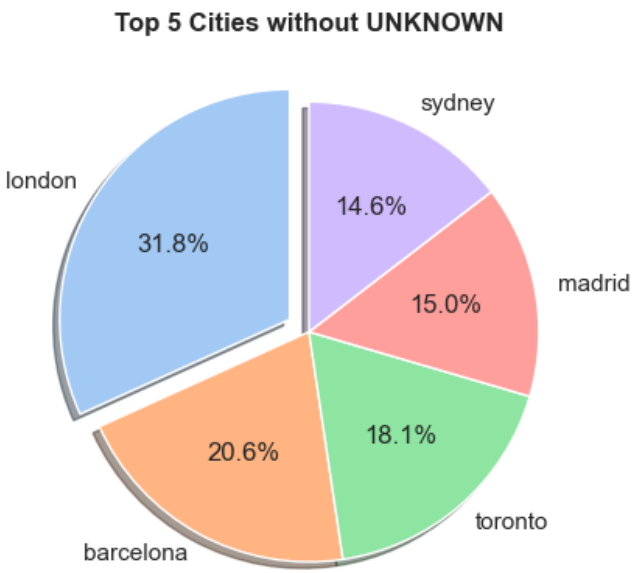


Figure 15: Piechar of cities of user's who gave ratings

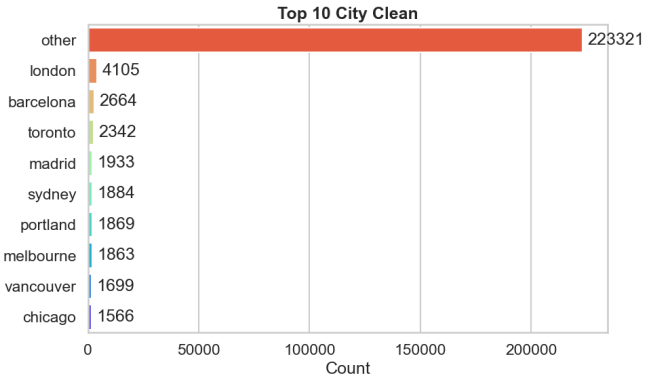


Figure 16: Barchart of cities of user's who gave ratings

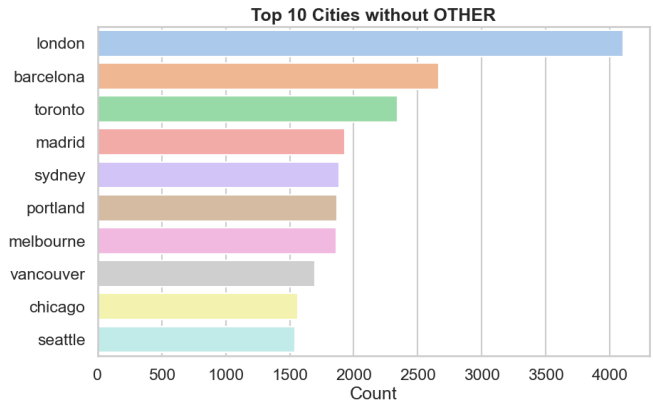


Figure 17: Barchart of cities of user's who gave ratings

The age column in our dataset is **not normally (Gaussian) distributed**. It is heavily skewed, with many extreme values, which makes standard outlier detection methods, such as the Interquartile Range (IQR), inadequate for this data. Which we can notice and compare with normal Gaussian distribution on following histograms:

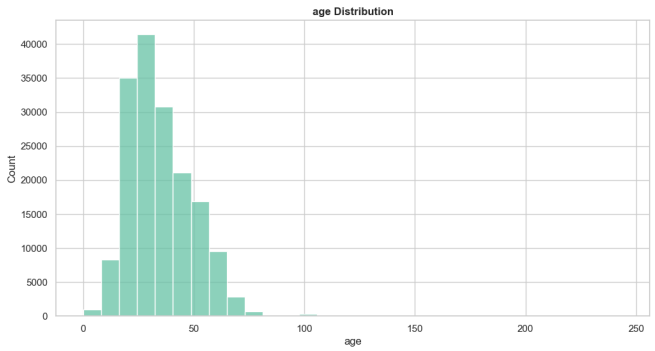


Figure 18: Age Histogram

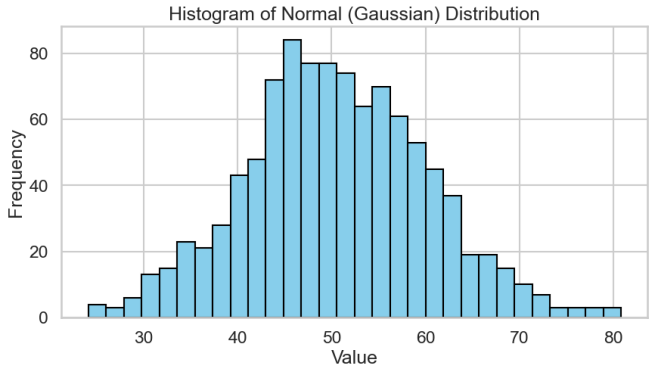


Figure 19: Normal Gaussian distribution histogram

Using the IQR method:

$$\text{IQR} = Q3 - Q1$$

- **Lower bound:** $Q1 - 1.5 \times \text{IQR} = 24 - 1.5 \times 30 = -6$ This is meaningless, as it would leave negative ages or zeros, which are invalid.
- **Upper bound:** $Q3 + 1.5 \times \text{IQR} = 44 + 1.5 \times 30 = 74$ This is too low, as it would incorrectly remove valid older users aged 75 99.

The IQR method alone is insufficient for this dataset. Instead, explicit filtering is required to ensure realistic and meaningful values. We choose to remove ages below 5 and above 99, which preserves the majority of valid data while removing unrealistic outliers.
The `age` column contains 168,096 entries with a mean of 34.75 and a standard deviation of 14.43. The values range from 0 to 244, with the 25th, 50th, and 75th percentiles at 24, 32, and 44, respectively. The standard deviation, σ, measures how spread out the values are around the mean and is calculated as

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2} \tag{13}$$

where x_i represents each individual age, μ is the mean age, and N is the total number of entries. In this dataset, the high value of $\sigma = 14.43$ reflects the presence of many extreme outliers, which is evident in boxplots where ages far beyond the typical range appear as points outside the whiskers.
For comparison, a standard normal (Gaussian) distribution with a mean around 50 and a standard deviation of 9.69 has values mostly concentrated between 17 and 84. Unlike this idealized normal distribution, our `age` data is heavily skewed.

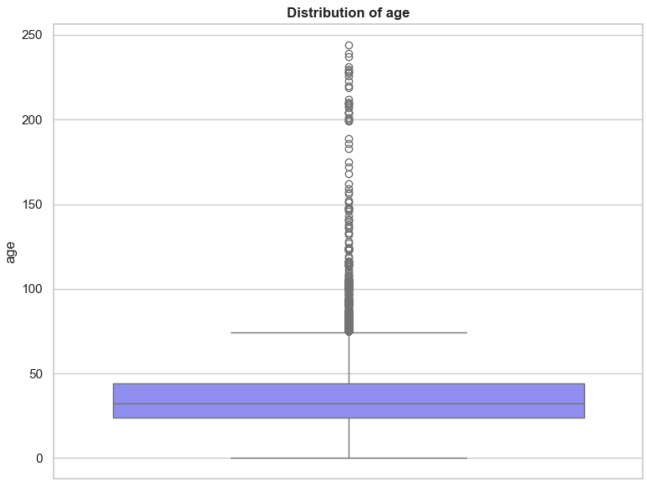


Figure 20: Age Boxplot

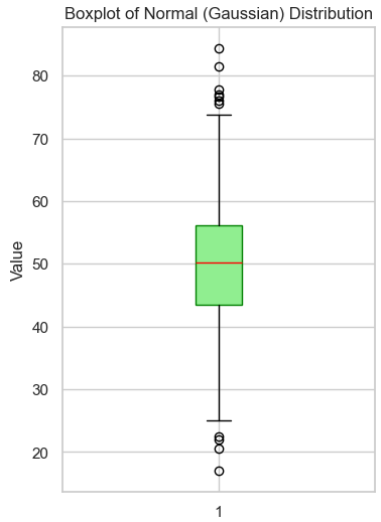


Figure 21: Normal Gaussian distribution Boxplot

After removing unrealistic ages below 5 and above 100, there are still a large number of missing values, with 112,043 rows lacking an `age`. Dropping these rows would cause significant data loss, so we impute them using a random method based on the existing age distribution. This approach uses the median to center the distribution, the standard deviation to preserve the spread, and ensures that generated ages stay within the realistic range of 5 to 100. The imputed values are rounded to integers, maintaining a discrete age variable. After imputation, the overall age distribution remains similar to the original data, no extreme outliers are introduced, and all ages fall within a plausible range. The final histogram and boxplot confirm that the distribution looks realistic.

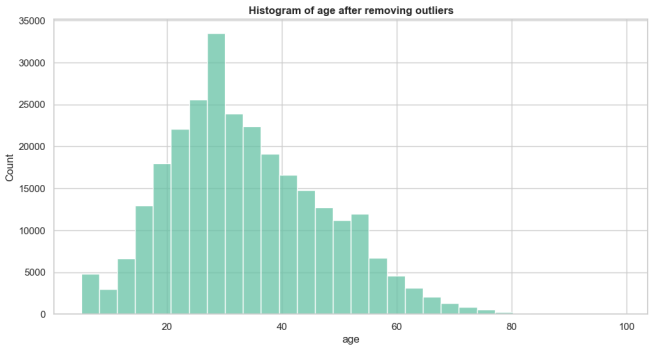


Figure 22: Age Histogram Final

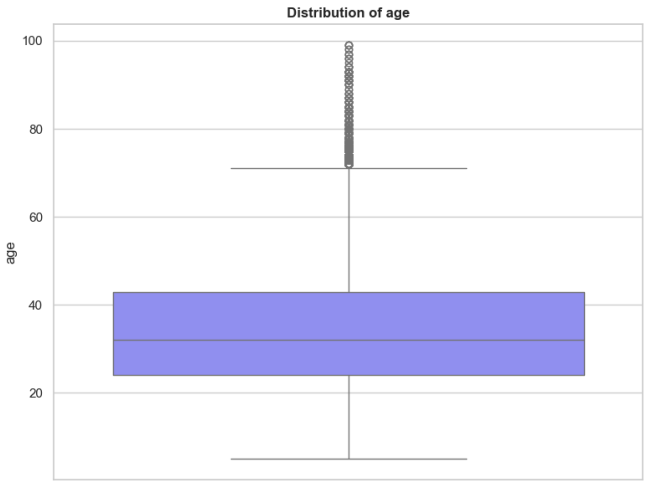


Figure 23: Age Boxplot Final

4.4 Merging datasets and Feature engineering

We merge our datasets to continue cleaning and enriching the data, allowing us to create new features and gain deeper insights. The `users_df` (278,858 rows, 6 columns) is merged with the `ratings_df` (1,149,780 rows, 3 columns) on the `user_id` column. Each resulting row represents a rating linked to the corresponding user's information.
By default, `pd.merge()` performs an inner join, which keeps only the `user_id`s that exist in both datasets. Users without ratings, or ratings without matching user information, are excluded from the result.
When merging with the books dataset, we join on the `isbn` column rather than `book_title`, since `isbn` uniquely identifies each book edition. In contrast, `book_title` is not unique, multiple editions may share the same title, and inconsistencies such as typos or casing differences can lead to duplicate rows, incorrect matches, or missing records. Joining on `isbn` ensures accuracy and consistency across datasets.

We can observe that the books with the most ratings and highest average scores include popular titles such as *The Da Vinci Code*, *Harry Potter and the Sorcerer's Stone*, *To Kill a Mockingbird*, and *The Lovely Bones*. The overall average rating across all books is around 4.2. However, as seen earlier, many users gave a rating of 0. These represent readers who read the book but did not actually rate it. After removing all 0 ratings, only two books have more than 500 valid ratings: *The Lovely Bones* by Alice Sebold, with an impressive average score of 8.18, and *Wild Animus* by Rich Shapero, with an average score of 4.39.

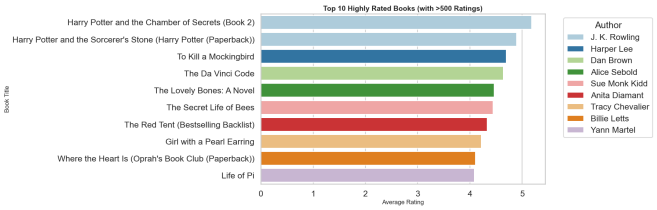


Figure 24: Most popular books with more than 500 ratings (0 ratings included)

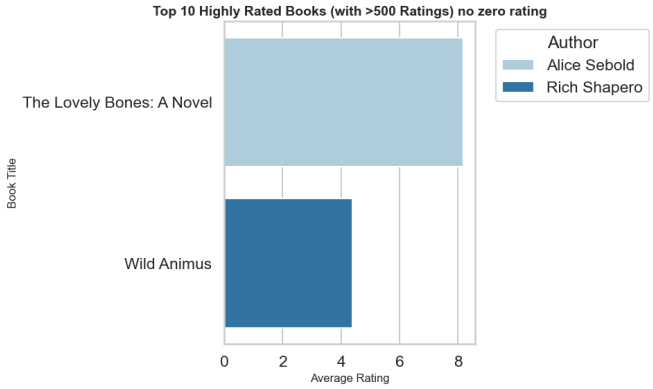


Figure 25: Most popular books with more than 500 ratings (NO 0 ratings included)

After removing all zero ratings, we can see which books received the most genuine user feedback. Among books with more than 300 ratings, popular titles such as *Harry Potter and the Sorcerer's Stone*, *The Da Vinci Code*, *Life of Pi*, and *The Lovely Bones* dominate the list, each with average scores above 8. These books are well-known bestsellers, suggesting a strong link between popularity and the number of user ratings.
When we raise the threshold to 400 ratings, only four books remain: *The Da Vinci Code* by Dan Brown, *The Lovely Bones* by Alice Sebold, *The Secret Life of Bees* by Sue Monk Kidd, and *Wild Animus* by Rich Shapero. Interestingly, while the first three books maintain high average ratings around 8, *Wild Animus* stands out with a much lower score of 4.39, indicating that a large number of ratings does not always correlate with positive feedback.

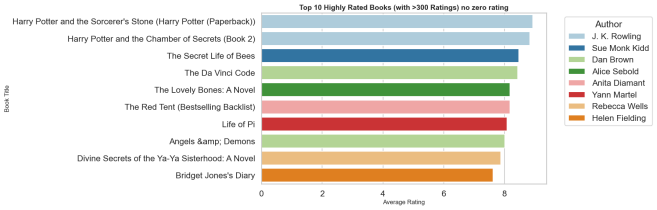


Figure 26: Most popular books with more than 300 ratings (NO 0 ratings included)

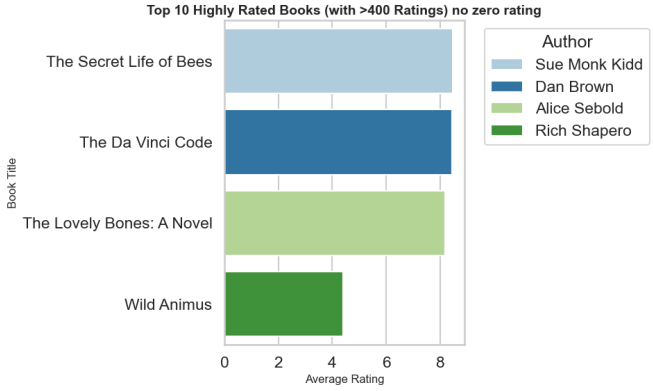


Figure 27: Most popular books with more than 400 ratings (NO 0 ratings included)

4.4.1 User Level Features

We generate user-level features to better understand our dataset and capture user behavior. These include the average rating each user gives (`user_avg_rating`), the total number of ratings per user (`user_num_ratings`), and the consistency of their ratings measured by the standard deviation (`User_rating_variability`). All these features are collected in a separate dataframe for easier analysis. Examining the results, we notice that most users rated only one book (39,223 users), with a rapid drop for two or three ratings, highlighting a significant cold start problem that challenges user-based recommendation approaches. Nonetheless, some users are highly active, rating hundreds of books, which can still provide valuable patterns for generating recommendations.

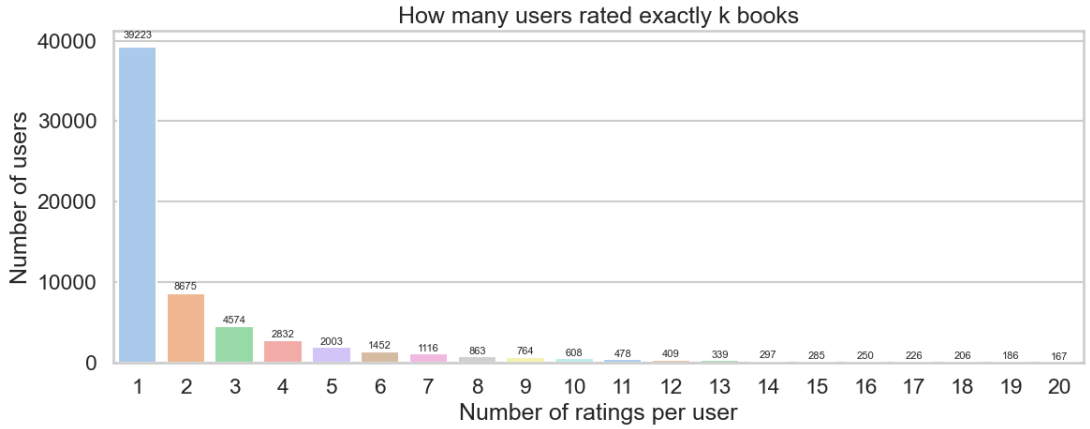


Figure 28: How many users rated K books

Also we can observe average ratings per user.

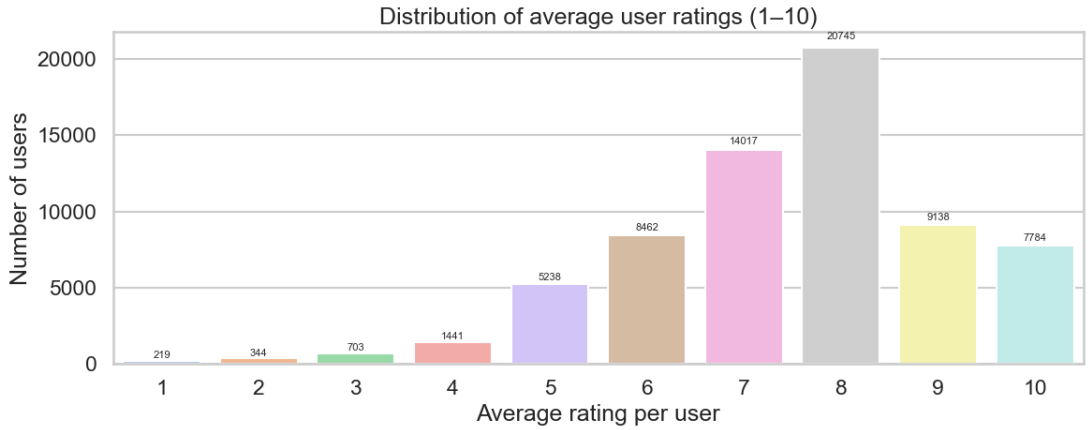


Figure 29: Average rating per user

We create a user age group feature by dividing ages into five ranges: 0–18, 18–25, 25–35, 35–50, and 50–100, and assigning numeric codes from 1 to 5. Since many users rate only one book and many books are rated by only one user, the standard deviation of ratings cannot be calculated for these cases. We replace these missing variability values with 0, indicating no variation in their ratings.

4.4.2 Book Level Features

We create book-level features to better understand and analyze our dataset. These include the average rating for each book (`book_avg_rating`), the total number of ratings per book (`book_num_ratings`), the variation in book ratings measured by standard deviation (`book_rating_variability`), and a popularity metric that combines average rating with the number of ratings (`book_popularity_score`). A summary dataframe is then generated containing these unique book-level features. Examining the number of ratings per book, most books were rated only once (88,137 books), with the count dropping sharply for two or three ratings. This highlights the sparsity of the dataset, which can make it harder for item-based collaborative filtering to find similar books. Nevertheless, some books received hundreds of ratings, with a few exceeding 500, showing that popular books can still provide strong signals for recommendations.

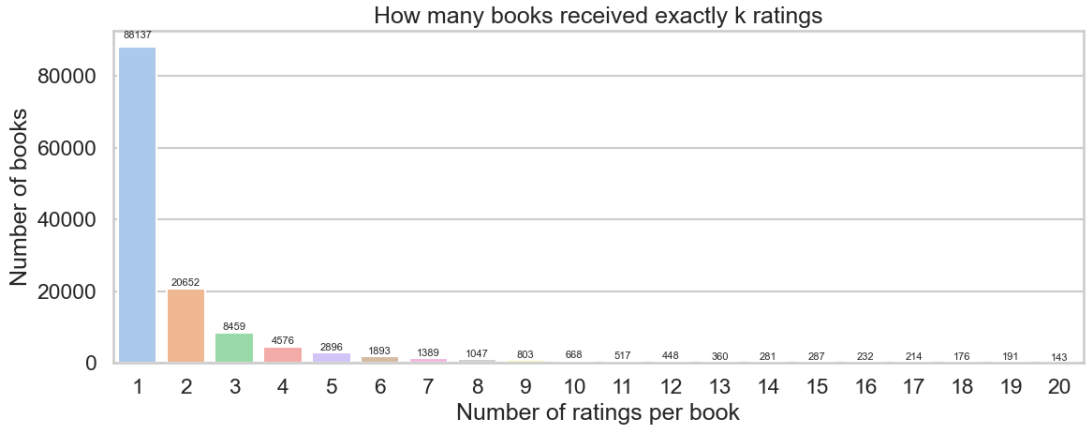


Figure 30: How many books recieved K books

We analyze the distribution of average ratings per book to see how many books received each specific rating. Most books cluster around high-range ratings, while a small amount of books achieve very low averages. This gives insight into the overall perception of books in the dataset and helps identify highly rated or unpopular books.

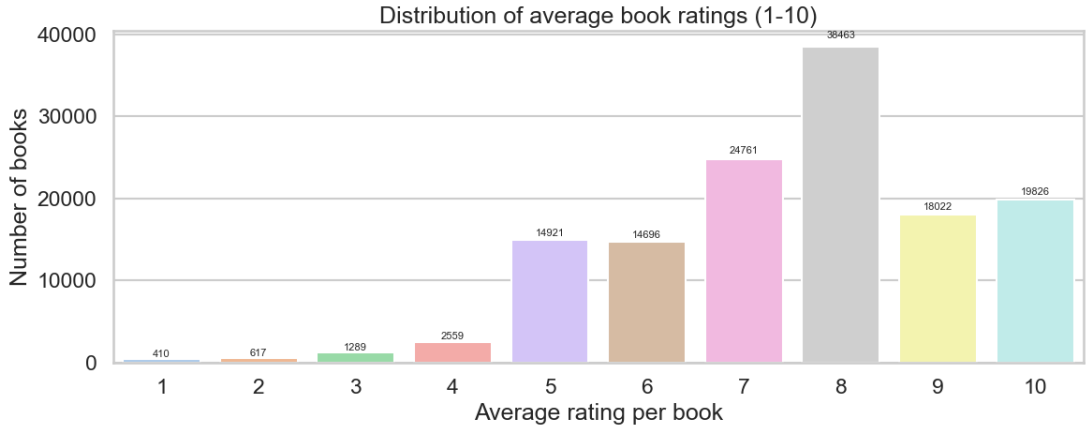


Figure 31: Books Average Rating Distribution

4.4.3 Location, Author, Publisher Level Features, Additional Plots and Insights

We can extract additional features to gain deeper insights from our dataset. This includes location based features such as the average rating per country (`country_avg_rating`), per region (`region_avg_rating`), and per city (`city_avg_rating`). We also capture personal bias with `user_country_rating_bias`, which measures the difference between a user's average rating and their country's average, and we track the local popularity of books using `book_country_avg_rating`, the average rating of each book within each country.
Additional features are created at different levels, including author level features such as `author_avg_rating`, the average rating of all books by each author, and publisher level features such as `publisher_avg_rating`, the average rating of all books by each publisher. We also compute the age of each book (`book_age`) as the difference between the current year (2025) and its year of publication, providing context for temporal analysis of ratings.

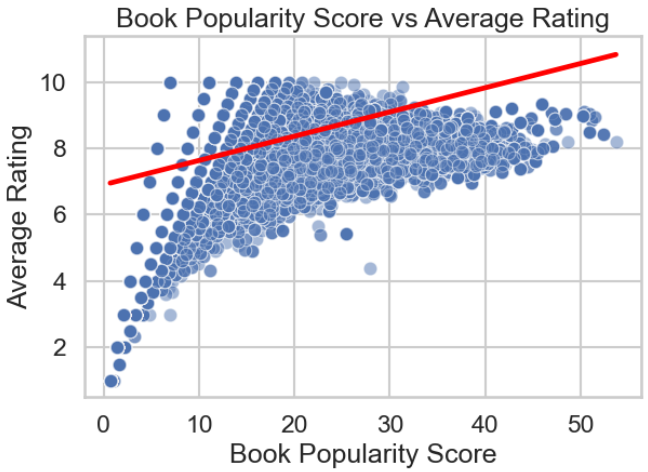


Figure 32: Books average rating and book popularity

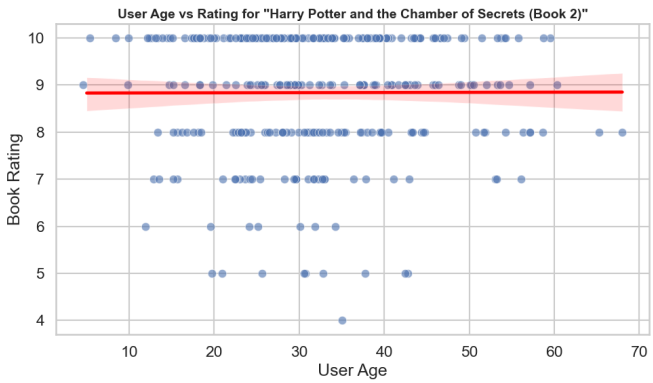


Figure 33: Harry Potter Ratings and User Age

Heatmaps showing correlation between authors, publishers and user ages.

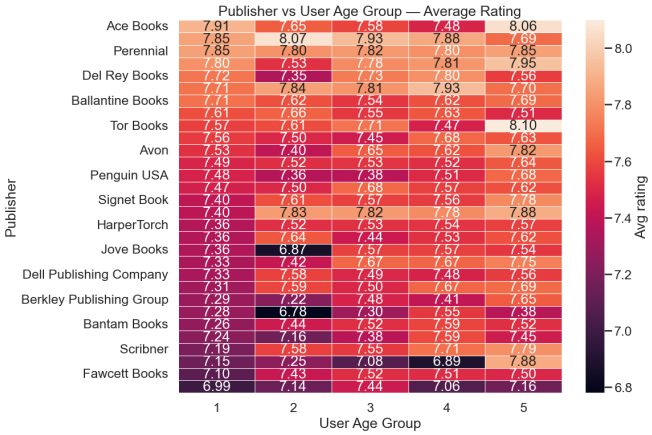


Figure 34: Publisher user age heatmap

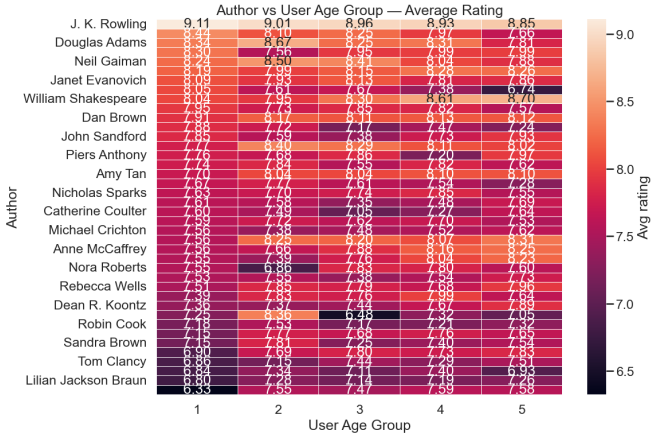


Figure 35: Author user age heatmap

5 Collaborative Filtering Models Building and Evaluation

As explained in detail in **collaborative_filter**, this section implements the theoretical concepts discussed earlier. For collaborative filtering, the model only requires the core useritem interactions: `user_id`, `isbn` or encoded book title, and `book_rating`. Other metadata, such as user demographics or book averages, are not needed in standard CF, although they can be incorporated in hybrid approaches.

CF works by capturing similarity patterns: users who rate similar books in similar ways, and books that are rated similarly by similar users. Columns like age, country, or `author_avg_rating` are considered side information. While these are not part of the core CF matrix, they can be useful for content based recommenders, hybrid models combining CF with features, or for deeper analytics and fairness evaluations.

For a pure CF implementation, only the useritemrating data is necessary. It is also important to filter out inactive users and books with very few ratings, since new users and items lack sufficient interaction history to generate meaningful recommendations.

To reduce the cold start problem, we first select users who have rated at least 10 books and books that have received at least 10 ratings. After this initial filtering, some users or books may fall below the threshold because removing books decreases the number of ratings for users, and removing users decreases the number of ratings for books. This cascading effect requires iterative filtering: we repeatedly recompute the counts and apply the thresholds until the dataset stabilizes. Initially, 6,589 users meet the threshold of 10 ratings, and 5,712 books have been rated by at least 10 users. After applying iterative filtering, we are left with 2,272 books and 2,508 users, as removing users and books successively eliminates some ratings for the remaining items.

After applying the iterative filtering, the distribution of user activity and book ratings is still extremely sparse. For users, only a small number have given many ratings: for example, 237 users rated exactly 10 books, 202 rated 11 books, 171 rated 12, and so on, with very few users contributing more than 100 ratings. Similarly, for books, only a limited number have received many ratings: 289 books have exactly 10 ratings, 261 have 11, 203 have 12, and the counts rapidly decline for higher numbers, with some books receiving over 100 ratings but very few reaching the upper range. This highlights that even after filtering, the dataset remains extremely sparse, making collaborative filtering challenging and emphasizing the importance of focusing on the most active users and popular books.

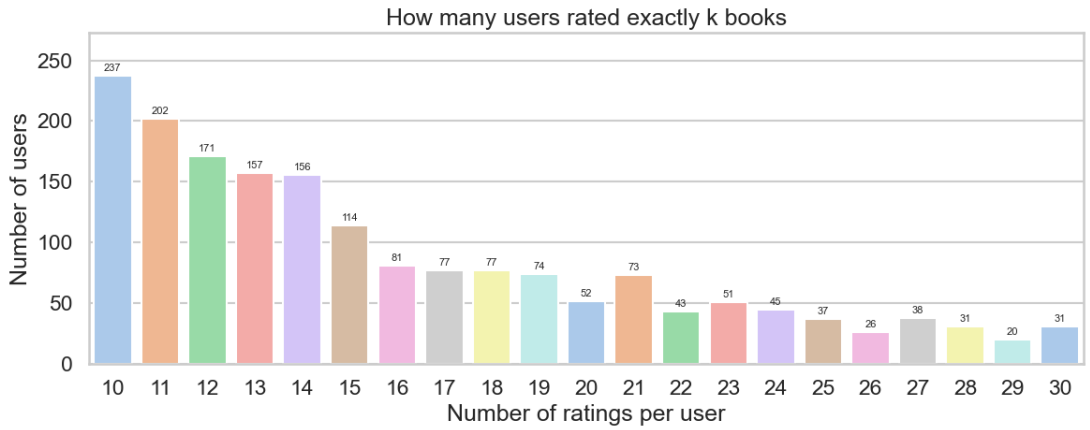


Figure 36: How many users have rated K books

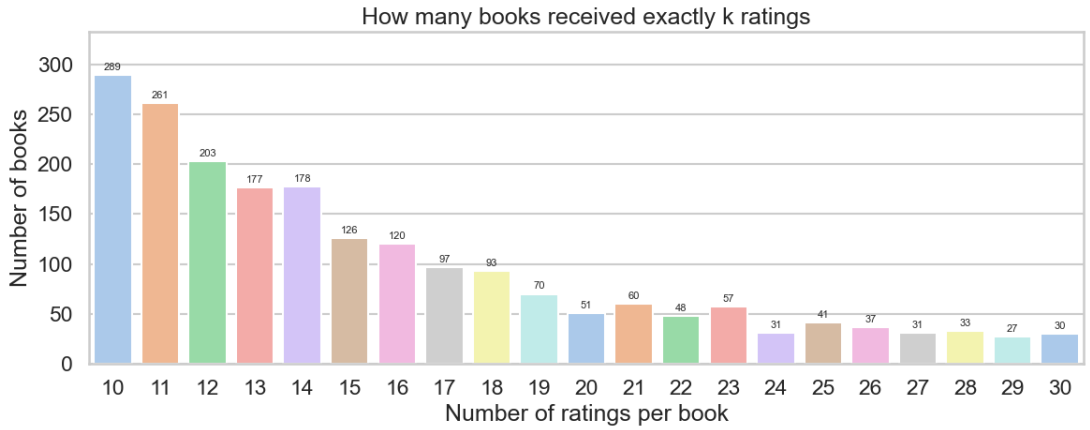


Figure 37: How many books have recieved K rating

We first set up the data split by reserving 20% of each user's ratings for testing, using a fixed random seed to ensure reproducibility. The dataframe is then grouped by user, and each user's ratings are shuffled randomly. For each user, the first portion of their ratings is assigned to the test set, with the remainder going to the training set. If a user ends up with only one rating in either set, one rating is moved to the training set to ensure every user appears in both sets. The resulting `train_df` contains all training ratings and `test_df` contains all test ratings, with indices reset for cleanliness. This procedure guarantees that all users are represented in both train and test sets, allowing collaborative filtering models to be trained and evaluated reliably. The final print statements confirm the number of rows and unique users in each split.

We encode the `user_id` and `book_title` columns into numeric indices because collaborative filtering models require a user–item matrix with integer row and column indices. String IDs cannot be used directly in most matrix-based algorithms.

Using **label encoding**, each unique user and book in the training set is assigned a distinct integer, stored in `user_idx` and `book_idx`. The same mapping is then applied to the test set to ensure consistency, so the same user or book always has the same index across both sets.

After encoding, the number of unique users and books is printed, which should match the counts in the training set. This prepares the data for constructing the interaction matrix, where rows correspond to users, columns correspond

to books, and entries represent ratings.
In short, this step transforms categorical IDs into integers, enabling the creation of the user–item matrix required for collaborative filtering.

5.1 User Based CF

We construct a sparse matrix $\mathbf{R} \in \mathbb{R}^{m \times n}$ using `csr_matrix`, where each entry R_{ui} represents the rating given by user u to book i. Rows correspond to users and columns correspond to books, defining the dimensions of the matrix. Using a sparse matrix is memory efficient, since most users have rated only a small fraction of all books, leaving the matrix mostly empty.
In collaborative filtering, each user can be represented as a vector of their ratings. The length of a user vector reflects the magnitude of their ratings: a user who gives many high ratings has a longer vector, while a user who gives few or low ratings has a shorter vector. Cosine similarity, however, ignores vector length and focuses on the angle between vectors, capturing how similar the pattern of ratings is between users. This allows identification of users with similar preferences across books, even if their overall rating tendencies differ. By using top-K neighbors, we consider only the most similar users when generating recommendations.

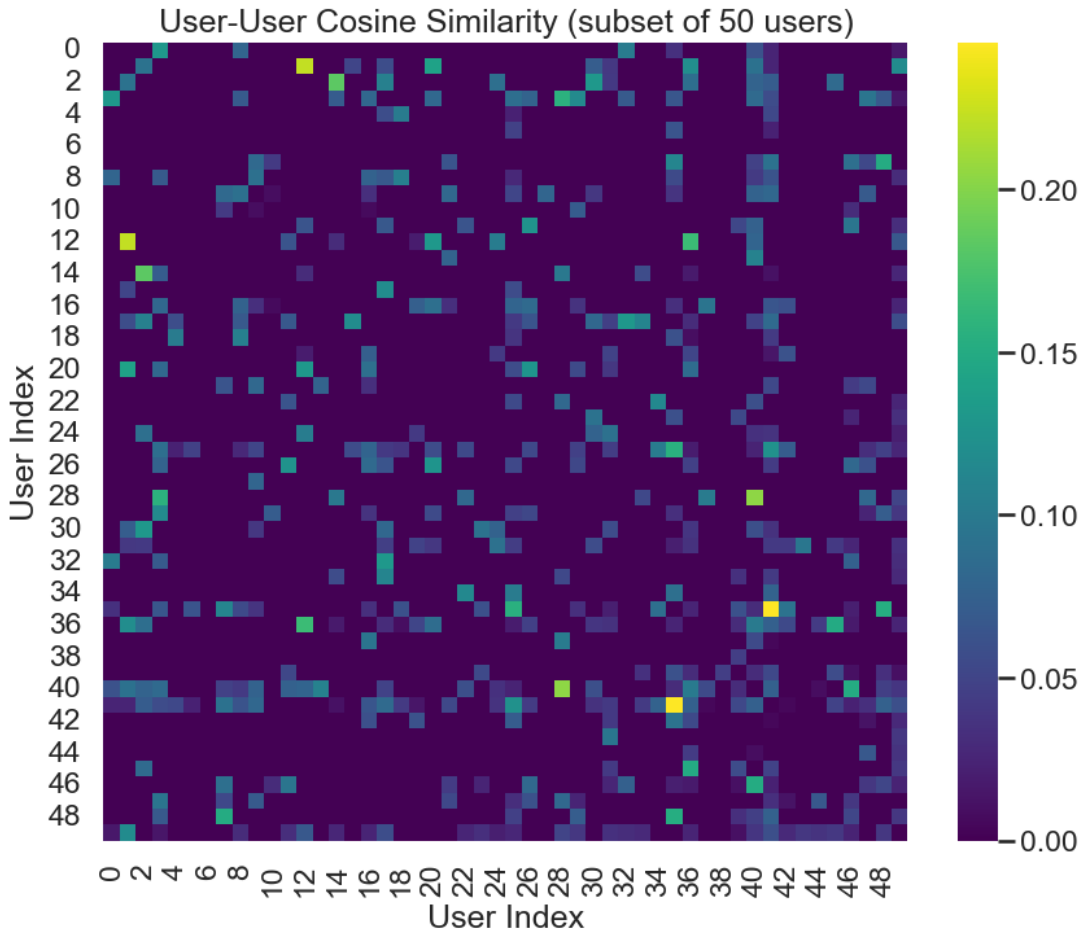


Figure 38: User User cosine similarity heatmap

Cosine similarity ranges from 0, indicating no similarity, to 1, indicating high similarity. In the heatmap, darker colors correspond to low similarity (closer to 0), while lighter colors indicate high similarity (closer to 1). Setting the diagonal to 0 removes self-similarity, ensuring that each user is not considered similar to themselves when identifying neighbors. For each user, we first identify the top-K most similar users from the user-user similarity matrix. These neighbors serve as the basis for generating recommendations, as their ratings provide insight into what the target user might like.
To predict ratings, we calculate a weighted average of the neighbors' ratings for each user-item pair, using the similarity scores between the target user and each neighbor as weights. The resulting weighted sum is then normalized by the

sum of similarity scores to ensure that the predicted ratings are scaled appropriately. If none of the neighbors has rated a given item, a fallback strategy can be used, such as the user's average rating or the global average rating across all users.
This process is repeated for every user and every item, producing a full prediction matrix. Each entry in this matrix represents the estimated rating that a particular user would give to a specific item, allowing the system to make personalized recommendations.

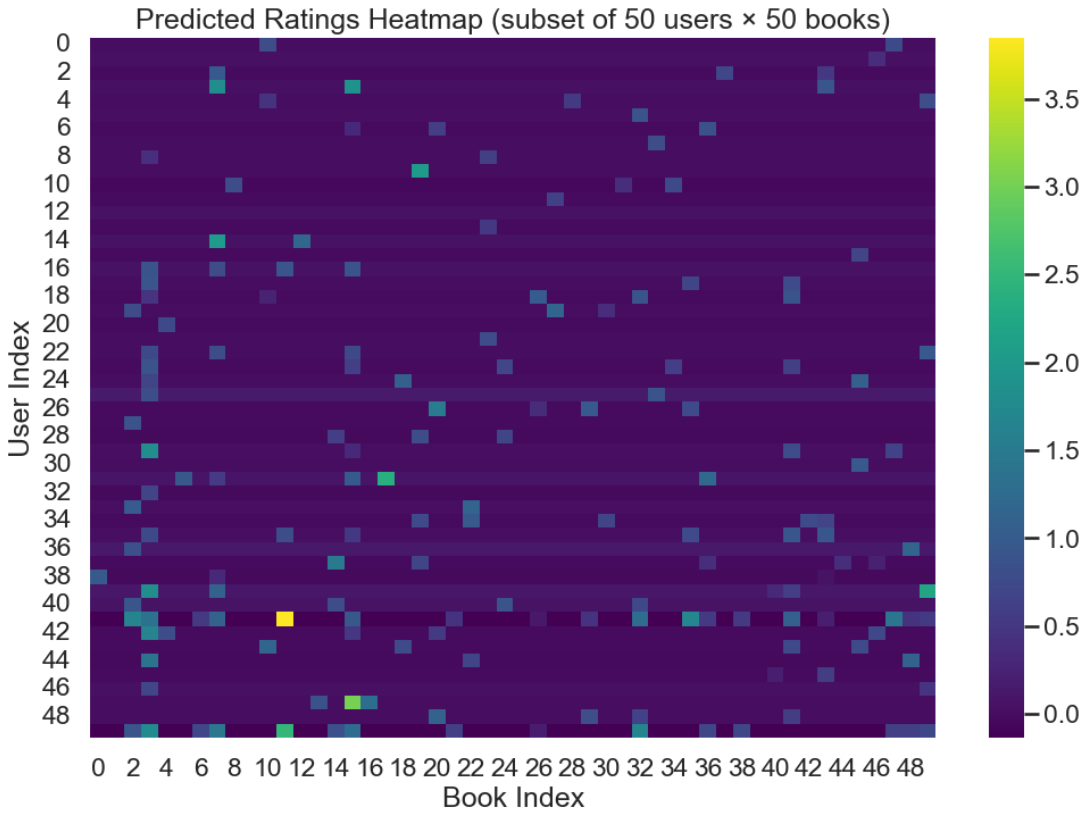


Figure 39: User Book rating prediction matrix

The evaluation metrics paint a brutally honest picture of the system's performance. Precision and recall indicate how well the system recommends books that users actually like, while MAE, MSE, and RMSE quantify how far the predicted ratings deviate from the true ratings.
Looking at the top-k metrics, Precision@5 is only 0.0542. This means that out of the five books the system predicts a user will like most, on average, only about one in twenty is actually relevant. Even when increasing the recommendations to 20, Recall rises only to 0.1116, so the system captures barely 11% of the items a user truly rated. Precision continues to drop as k increases, indicating that recommending more items mostly adds irrelevant books. This performance is painfully low, though not surprising given the extreme sparsity typical of book rating datasets.
The rating prediction errors are equally harsh. A mean absolute error (MAE) of 7.3627 and a root mean squared error (RMSE) of 7.6219 mean that, on average, the predicted rating is off by more than 7 points on a typical 1–10 scale. The mean squared error (MSE) of 58.0931 highlights that some predictions are catastrophically wrong. In practice, this suggests that the model is essentially guessing most of the time. While it may capture very rough trends, it is **terrible at predicting individual ratings** and fails to produce meaningful recommendations for real users.
In short, this model generates numbers but is nowhere near useful in practice. If deployed as-is, users would receive mostly irrelevant recommendations.

5.2 Item Based CF

To implement item-based collaborative filtering, we first transpose the original user-item rating matrix. In the user-item matrix, each row corresponds to a user and each column corresponds to a book. By transposing the matrix, we switch the axes so that each row now represents a book and each column represents a user.

This transformation allows each book to be treated as a vector in the space of users, where the entries correspond to the ratings given by all users. Computing similarity between books then becomes a matter of comparing these vectors, typically using cosine similarity or another similarity metric.
Formally, if $\mathbf{R} \in \mathbb{R}^{m \times n}$ is the original user-item matrix with m users and n books, we construct the item-user matrix $\mathbf{R}^\top \in \mathbb{R}^{n \times m}$ by taking the transpose. Each row $\mathbf{r}_i$ of $\mathbf{R}^\top$ is the rating vector for book i, allowing us to compute similarities between books and generate recommendations based on item-to-item relationships.

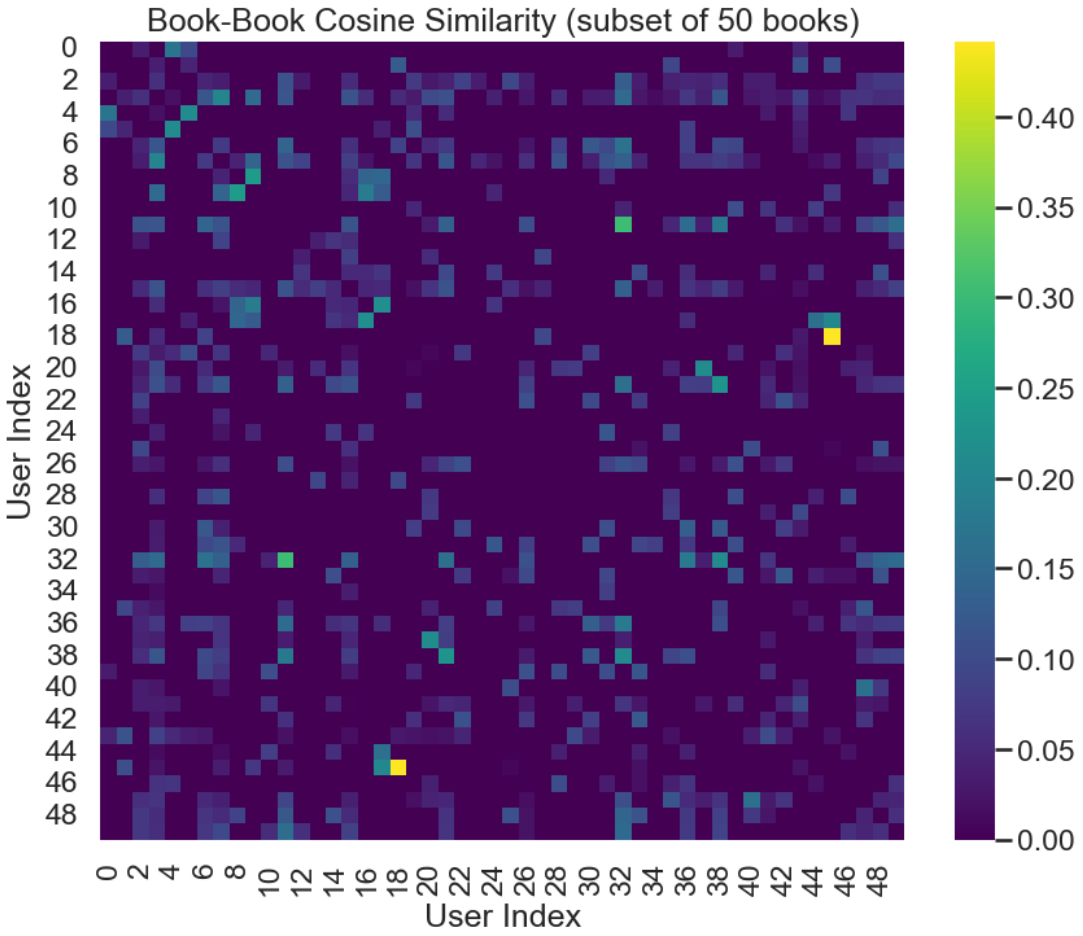


Figure 40: Item Item cosine similarity heatmap

Once the item-user matrix is constructed, we compute the similarity between books using cosine similarity. This results in an item-item similarity matrix, where each entry quantifies how similar two books are based on the ratings they received from all users.
For each book, we then identify the top-K most similar books according to the similarity scores. These neighbors are used to generate predicted ratings for all users: for a given user and a target book, the predicted rating is calculated as a weighted average of the ratings that the user has given to the top-K most similar books. The weights correspond to the similarity scores between the target book and each neighbor.
Repeating this process for every book and every user produces a full predicted rating matrix. Each entry in this matrix represents the estimated rating a particular user would give to a specific book, allowing the system to generate personalized recommendations based on item-to-item similarity rather than user-to-user similarity.

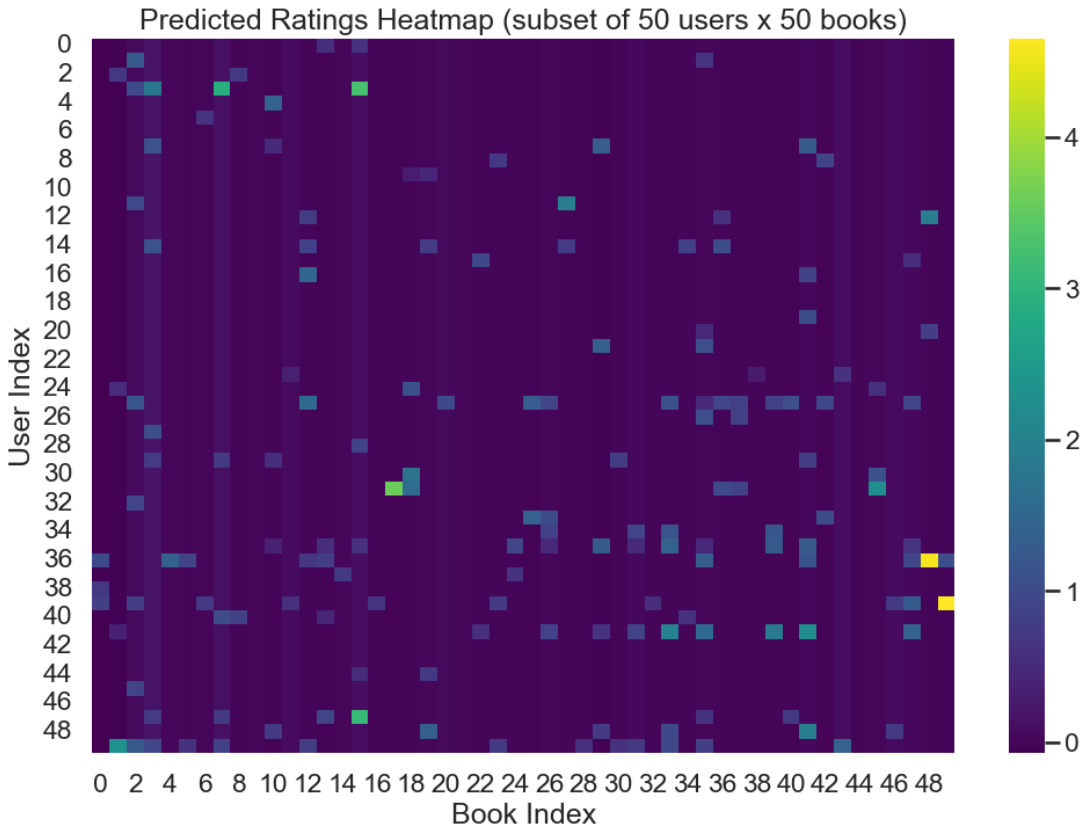


Figure 41: Item based prediction atrix heatmap

Examining the top-N metrics, Precision@5 is only 0.0742, meaning that out of the five books recommended to a user, **less than one book on average is actually relevant**. Even when N is increased to 20, the system captures only about 16% of the relevant items in Recall@20. Precision decreases as more items are recommended, showing that most suggestions are irrelevant. This modest improvement over the user-based method indicates that item-based collaborative filtering captures some patterns in item similarity, but it is still extremely weak for meaningful recommendations. Sparse user ratings and limited overlap between items are major limiting factors.
The rating prediction errors remain severe. A mean absolute error (MAE) of 7.1474 and a root mean squared error (RMSE) of 7.4423 indicate that, on average, the predicted ratings are **off by more than 7 points**, which is substantial on a typical 1–10 scale. The mean squared error (MSE) of 55.3873 highlights that some predictions are catastrophically wrong. While the item-based approach slightly improves ranking of relevant items compared to user-based collaborative filtering, it still **fails to predict actual ratings accurately** and would produce mostly useless recommendations if deployed.
In summary, item-based collaborative filtering is only a small step forward. Most recommendations remain irrelevant, and predicted ratings are wildly inaccurate, making this method far from practical for real-world usage.

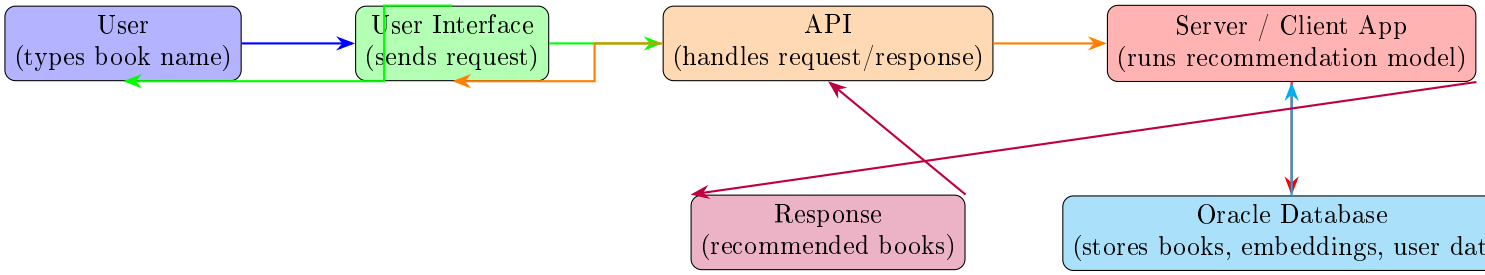
6 Deployment of Book Recommendation System

Having an app on our local system simply isn't enough, it needs to be accessible online. Deployment is the process of making your application available for users so they can access and interact with it. It usually means moving your code from a development environment to a live production environment, where the application is fully operational.

1. **User Input:** The user types a book title (and optionally a user ID) in the application interface.
2. **User Interface / Front END:** The user using frontent sends a request and formats it into an API request (e.g., JSON) that is sent to the API.
3. **API:** The API acts as a gateway. It receives the request, validates it, and forwards it to the server. It also ensures that only necessary information is exchanged, keeping internal details secure.
4. **Server / Client APP / Backend:** The server runs the recommendation logic:

- Looks up the book embedding from the Oracle database or computes it if it's free text.
- Performs a nearest-neighbor search in the embedding space to find similar books.
- Optionally combines item similarity with user profile data for personalization.

5. **Oracle Database:** Stores all persistent data, including books, embeddings, and user interaction history. The server queries it using SQL, retrieves the necessary data, and processes it in memory.
6. **Response:** After computing the Top-N recommended books, the server sends the results back to the API, which forwards them to the client app. The user sees the recommendations seamlessly in the interface.



- **User → User Interface → API → Server/Client App → Database:** This is the request path.
- **Database → Server/Client App → API → User Interface → User:** This is the response path.
- The API acts as a secure bridge and ensures that internal server and database logic is not exposed to the client or user.
- The Oracle database stores all persistent data, and the server retrieves only what is needed for computing recommendations.
- The server performs the actual recommendation computation, including vector lookups, nearest-neighbor searches, and optional personalization.

6.1 Application Programming Interface (API)

An API, short for Application Programming Interface, is basically a bridge that lets different software programs talk to each other. It defines a set of rules and protocols that allow applications to exchange data, features, or functionality. Instead of building every feature from scratch, developers can use APIs to connect their apps with existing services or data sources, saving time and effort. APIs also give application owners a controlled and secure way to share certain parts of their app's data or capabilities—either with internal teams or with outside users. They're designed to expose only the specific information or actions that are needed for a given task, not the entire system. By doing this, APIs help protect sensitive information and maintain overall system security while still enabling smooth, efficient communication between different software systems.

It helps to think of API communication as a conversation between a client and a server. In a book recommender system, the user's app—the client—sends a request, and the server responds with recommendations. The API is the bridge that makes this connection possible.

Here's how it works in practice: imagine a user types in the name of a book they like. When they hit "Submit," the app sends this information as a request through the API. The request travels to the server, which runs a trained recommendation model to find books similar to the one the user entered.

The server then sends back a response containing a list of recommended books. The API takes this data and delivers it to the app, so the user sees their personalized recommendations almost instantly.

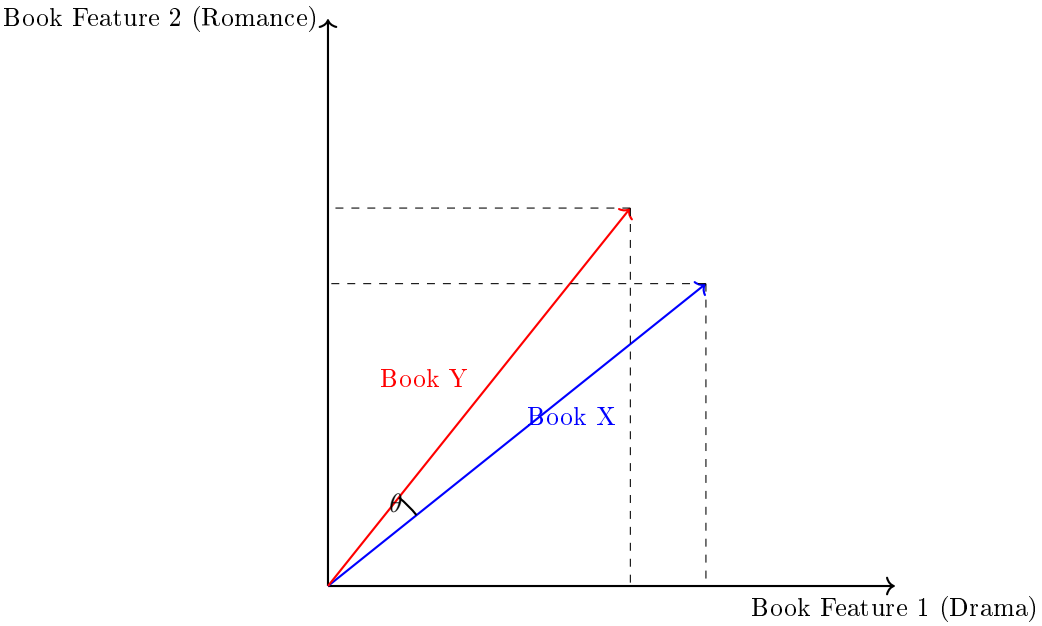
From the user's perspective, it feels seamless—they just type a book name and get suggestions. Behind the scenes, though, the API is carefully handling the request and response, sharing only the information needed, keeping the system secure, and making sure the app and server communicate smoothly.

7 Similarity/Distance Measures

In content-based filtering, items are compared using their feature representations, and those that are closer in feature space are considered more similar. In collaborative filtering, similarity is computed between users or items based on their interaction or rating patterns, with closer vectors indicating stronger similarity. Both approaches use metrics like cosine similarity or correlation to quantify closeness.

7.1 Cosine Similarity

One of the most common methods, especially for high-dimensional feature spaces, is **Cosine Similarity**. Cosine similarity measures cosine of the angle between two vectors, giving a value between -1 and 1. The closer the value is to 1, the more similar the items are considered.



Here, θ is the angle between vectors $\mathbf{x}$ and $\mathbf{y}$. Cosine similarity measures how aligned these vectors are: the smaller the angle, the closer the similarity is to 1.
Intuition: Cosine similarity focuses on the direction of vectors rather than their magnitude. Two items with similar feature patterns are considered similar, even if one has generally higher values than the other.
The formula for cosine similarity between two item vectors $\mathbf{x}$ and $\mathbf{y}$ is:

$$\text{sim}(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\| \, \|\mathbf{y}\|} = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2} \, \sqrt{\sum_{i=1}^{n} y_i^2}} \tag{14}$$

Here, $\mathbf{x} \cdot \mathbf{y}$ represents the dot product of the two vectors, where $\mathbf{x}$ and $\mathbf{y}$ are vectors, for example, item feature vectors in a recommendation system and $\|\mathbf{x}\|$ and $\|\mathbf{y}\|$ are the magnitudes (lengths) of the vectors. This metric allows the system to quantify similarity between items and recommend those that are most closely aligned with the user's previous preferences.
In content-based filtering, each item is represented as a vector in a multi-dimensional feature space. For example, a book could be represented as

$$\mathbf{x} = [\text{drama score}, \text{romance score}, \text{adventure score}, \ldots].$$

- The **dot product** of two vectors $\mathbf{x}$ and $\mathbf{y}$ quantifies how much the vectors point in the same direction:

$$\mathbf{x} \cdot \mathbf{y} = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n = \sum_{i=1}^{n} x_i y_i \tag{15}$$

  Each term represents the contribution of a feature to the overall similarity.

- The **norm** or length of a vector is:

$$\|\mathbf{x}\| = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2} \tag{16}$$

  Normalizing by the vector lengths ensures that cosine similarity measures direction rather than magnitude.

Cosine similarity is then defined as:

$$\text{sim}(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\| \, \|\mathbf{y}\|} = \cos\theta \tag{17}$$

where θ is the angle between the vectors. The similarity ranges from -1 (opposite) to 1 (identical), with 0 indicating orthogonal (no similarity).

7.1.1 Euclidean Distance

Euclidean distance is a way to measure how far apart two items are in a multi-dimensional feature space. It is the length of the straight line connecting the two points representing the items. In recommendation systems, a smaller Euclidean distance between two items indicates that they are more similar in terms of their features.
Suppose we represent two books as feature vectors:

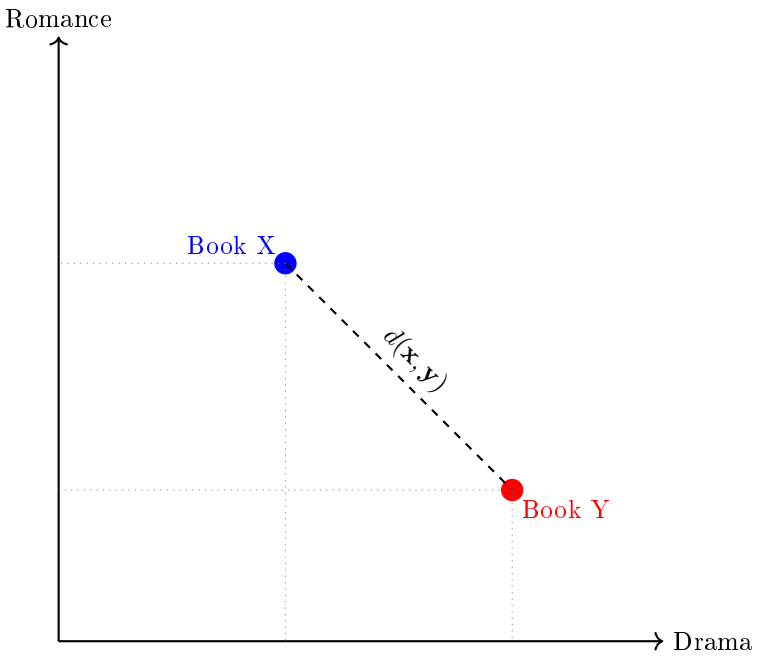
$$\mathbf{x} = (\text{drama score}, \text{romance score}, \dots), \quad \mathbf{y} = (\text{drama score}, \text{romance score}, \dots)$$

For n features, the Euclidean distance between $\mathbf{x}$ and $\mathbf{y}$ is:

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{(\text{drama}_x - \text{drama}_y)^2 + (\text{romance}_x - \text{romance}_y)^2 + \dots} = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \tag{18}$$

- Each term $(x_i - y_i)^2$ measures the squared difference in a single feature (e.g., drama, romance) between the two books.
- Summing over all features gives a combined measure of difference across all dimensions.
- Taking the square root converts this sum into the actual straight-line distance.

Intuition: - Two books with similar ratings across all features will have a **small Euclidean distance**. - Books with very different ratings in one or more features will have a **larger distance**.



This diagram shows:

- Each book as a point in a 2D feature space (Drama vs Romance).
- The dashed line representing the Euclidean distance between the books.
- Dotted lines showing the projections to each feature axis for clarity.

8 Evaluation Metrics

To assess the performance of the content-based recommender system, we employ two categories of evaluation metrics: (1) prediction accuracy metrics, which measure how accurately the system predicts user ratings, and (2) top-N recommendation metrics, which evaluate the quality of the ranked lists of recommended books.

8.1 Prediction Accuracy Metrics

These metrics evaluate how close the predicted ratings $\hat{r}_{ui}$ are to the actual user ratings r_{ui}.

- **Mean Absolute Error (MAE)**

$$MAE = \frac{1}{|T|} \sum_{(u,i) \in T} |r_{ui} - \hat{r}_{ui}| \tag{19}$$

  where T denotes the test set of user–item pairs. Lower MAE indicates higher predictive accuracy.

- **Root Mean Squared Error (RMSE)**

$$RMSE = \sqrt{\frac{1}{|T|} \sum_{(u,i) \in T} (r_{ui} - \hat{r}_{ui})^2} \tag{20}$$

  RMSE penalizes larger errors more heavily than MAE, and thus reflects how severely the system deviates from the actual ratings.

- **Mean Squared Error (MSE)**

$$MSE = \frac{1}{|T|} \sum_{(u,i) \in T} (r_{ui} - \hat{r}_{ui})^2 \tag{21}$$

  MSE serves as the base metric for RMSE, providing a measure of the average squared deviation between predicted and true ratings.

- **Normalized Mean Absolute Error (NMAE)**

$$NMAE = \frac{MAE}{r_{\max} - r_{\min}} \tag{22}$$

  NMAE normalizes the MAE by the range of the rating scale, enabling comparability across datasets with different rating intervals.

8.2 Top-N Recommendation Metrics

When the system produces a ranked list of recommended items, ranking-based metrics are preferred over pure rating prediction metrics.

- **Precision@N**

$$Precision@N = \frac{\text{Number of relevant items in top } N}{N} \tag{23}$$

  Precision@N measures the proportion of relevant items among the top-N recommendations.

- **Recall@N**

$$Recall@N = \frac{\text{Number of relevant items in top } N}{\text{Total number of relevant items}} \tag{24}$$

  Recall@N evaluates how well the system retrieves all relevant items for a user, indicating the completeness of recommendations.